W. DANIEL HILLIS

Mașina care gîndește

CUM FUNCȚIONEAZĂ CALCULATOARELE

HUMANITAS

SCIENCE MASTERS

W. DANIEL HILLIS este unul dintre informaticienii de vîrf cei mai activi ai zilelor noastre și a proiectat calculatorul cel mai rapid din lume. Este membru fondator și cercetător principal la Thinking Machines Corporation, unde a condus cercetările pentru cel mai important produs al companiei, The Connection Machine. Este editor al mai multor reviste științifice, între care *Artificial life*, *Complexity*, *Complex Systems* și *Future Generation Computer Systems*.

W. DANIEL HILLIS

MAȘINA CARE GÎNDEȘTE

Cum funcționează calculatoarele

Traducere din engleză de
MIHAI CIPU

HUMANITAS
BUCUREȘTI

Coperta
IOANA DRAGOMIRESCU MARDARE

Descrierea CIP a Bibliotecii Naţionale
HILLIS, DANIEL W.
Maşina care gîndeşte: cum funcţionează calculatoarele /
W. Daniel Hillis; trad.: Mihai Cipu. – București: Humanitas, 2001
192 p.; 20 cm – (Science Masters)
ISBN 973-50-0141-1

I. Cipu, Mihai (trad.)

004

W. DANIEL HILLIS
THE PATTERN ON THE STONE. THE SIMPLE IDEAS THAT MAKE COMPUTERS WORK
Orion Publishing Group Ltd.
© 1998 W.D. Hillis
Numele și marca „Science Masters“ sînt proprietatea lui Brockman, Inc.

© HUMANITAS, 2001, pentru prezenta versiune românească

ISBN 973-50-0141-1

PREFAȚĂ: MAGIE ÎN PIATRĂ

Scrijelesc un model geometric pe o piatră. Celui neinițiat, formele îi par complexe și misterioase, dar eu știu că, aranjate corect, ele vor da pietrei puteri speciale, făcînd-o capabilă să răspundă la descîntece într-o limbă pe care nicicînd ființele umane nu au vorbit-o. Voi pune pietrei întrebări în acea limbă, și ea îmi va răspunde arătîndu-mi o viziune: o lume creată de vraja mea, o lume imaginată în modelul de pe piatră.

Doar cu cîteva sute de ani în urmă, în New England-ul meu natal, o descriere precisă a ocupației mele ar fi făcut să fiu ars pe rug. Și totuși munca mea nu are nimic de-a face cu vrăjitoria; eu proiectez și programez calculatoare. Piatra este grăuntele de siliciu, iar descîntecele sînt programele de calculator. Modelele scrijelite pe grăuntele de siliciu și programele cu instrucțiuni pentru calculator pot părea complicate și misterioase, dar ele sînt produse în conformitate cu cîteva principii fundamentale, ușor de explicat.

Calculatoarele sînt cele mai complexe obiecte pe care noi, ființele umane, le-am creat vreodată, dar în esență sînt remarcabil de simple. Lucrînd în echipe cu doar cîteva zeci de oameni, am proiectat și construit calculatoare conținînd miliarde de componente active.

Diagrama circuitelor unei astfel de maşini, dacă ar fi vreodată desenată, ar umple toate volumele unei biblioteci publice de mărime mijlocie şi nimeni nu ar avea răbdarea să îşi arunce ochii peste toate. Din fericire, această diagramă e inutilă datorită regularităţii unui proiect de calculator. Calculatoarele sînt constituite dintr-o ierarhie de componente strict ierarhizate, fiecare componentă fiind repetată de multe ori. Tot ceea ce se cere pentru a înţelege un calculator este înţelegerea acestei ierarhii.

Un alt principiu ce face calculatoarele uşor de înţeles este natura interacţiunilor dintre componente. Aceste interacţiuni sînt simple şi bine definite. De obicei ele sînt unidirecţionale, încît acţiunile unui calculator pot fi clasificate cu claritate în cauze sau efecte, făcînd funcţionarea unui calculator mult mai uşor de înţeles decît, să spunem, funcţionarea internă a unui motor auto sau a unui aparat de radio. Un calculator are mult mai multe piese decît un automobil sau un aparat de radio, dar e mult mai simplu modul în care părţile lucrează împreună. Un calculator nu este dependent atît de mult de tehnologie cît de idei.

În plus, ideile nu au aproape nimic în comun cu componentele electronice din care sînt construite calculatoarele. În zilele noastre calculatoarele sînt construite din fire şi tranzistoare, dar ar putea fi construite la fel de bine, pe baza aceloraşi principii, din conducte şi valve sau din bastoane şi sfori. Principiile sînt esenţa a ceea ce face să funcţioneze un calculator. Unul dintre cele mai remarcabile lucruri legate de calculatoare este acela că principiile lor fundamentale transcend tehnologia. Subiectul cărţii de faţă sînt aceste principii.

Aceasta este cartea pe care aş fi dorit să o fi citit cînd am început să învăţ despre calcul. Spre deosebire de

cele mai multe cărți despre calculatoare — care descriu fie modul lor de utilizare, fie tehnologia cu care au fost construite (ROM, RAM, unități de disc etc.) — această carte este despre *idei*. Ea explică sau cel puțin prezintă majoritatea ideilor importante din informatică, inclusiv logica booleană, automatele cu un număr finit de stări, limbajele de programare, compilatoarele și interpretoarele, universalitatea în sensul lui Turing, teoria informației, algoritmica și complexitatea algoritmică, euristica, funcțiile logice, calculul paralel, calculul cuantic, rețelele neuronale, învățarea automată și sistemele cu autoreglare. Oricine este interesat de calculatoare în suficientă măsură pentru a citi această carte probabil a întîlnit deja multe dintre aceste idei, dar, cu excepția specializării în informatică, puține sînt ocaziile de a vedea cum se leagă între ele. Această carte prezintă conexiunile — întregul drum de la procese fizice simple, cum ar fi închiderea unui comutator, pînă la învățarea și adaptarea prezente la calculatoarele paralele cu autoreglare.

La baza unei prezentări a naturii calculatoarelor se află cîteva teme generale: prima este principiul *abstractizării funcționale*, care conduce la ierarhia, deja menționată, de cauze și efecte. Structura unui calculator este un exemplu de aplicare a acestui principiu — în repetate rînduri, la multe niveluri. Calculatoarele pot fi înțelese pentru că te poți concentra asupra a ceea ce se întîmplă la un nivel ierarhic fără a fi tulburat de detaliile a ceea ce se petrece la nivelurile inferioare. Abstractizarea funcțională este cea care separă ideile de tehnologie.

A doua temă unificatoare este principiul *calculatorului universal* — ideea că de fapt există un singur tip de calculator, sau, mai precis, că toate tipurile de cal-

culatoare sînt asemănătoare în privința a ceea ce pot și nu pot face. Din cîte putem spune, orice dispozitiv de calcul, fie el construit din tranzistoare, din tije și sfori, ori din neuroni, poate fi simulat de un calculator universal. Aceasta este o ipoteză remarcabilă: cum voi explica, ea sugerează că a face un calculator să gîndească precum creierul este doar o chestiune de programare corectă.

A treia temă în această carte, care nu va fi abordată pe deplin decît în ultimul capitol, este într-un fel antiteza primei teme. Ar putea exista un mod complet nou de a proiecta și programa calculatoarele, un mod care nu este bazat pe metodele standard din inginerie. Acest lucru ar fi senzațional, deoarece modul obișnuit în care noi proiectăm sistemele începe să eșueze cînd sistemele devin prea complicate. Înseși principiile ce ne fac capabili să proiectăm calculatoare conduc în cele din urmă la o anumită fragilitate și ineficiență. Acest punct slab nu are nimic de-a face cu vreo limită fundamentală a mașinilor de prelucrare a informației — este o limită datorată metodei ierarhice de proiectare. Dar dacă am folosi un proces de proiectare analog evoluției biologice — adică un proces în care comportarea sistemului *reiese* din acumularea multor interacțiuni simple, fără control „de sus în jos“ ? Un dispozitiv de calcul proiectat de un asemenea proces evolutiv ar putea prezenta ceva din robustețea și flexibilitatea unui organism biologic — cel puțin, așa se speră. Această abordare încă nu este înțeleasă pe deplin și s-ar putea să se dovedească nepractică. Ea este subiectul cercetărilor mele actuale.

În explicarea naturii calculatoarelor sînt cîteva cunoștințe de bază cu care trebuie să ne familiarizăm înainte de a trece la partea esențială. În primele două

capitole sînt prezentate elementele fundamentale: logica booleană, biții, automatele cu un număr finit de stări. Răsplata este că la sfîrșitul capitolului 3 veți înțelege cum funcționează calculatoarele, de sus pînă jos. Aceasta pregătește terenul pentru ideile captivante privind mașinile de calcul universale, a căror expunere începe în capitolul 4.

Filozoful Gregory Bateson a definit cîndva informația ca „deosebirea ce permite distingerea". Altfel spus, informația constă în diferențele pe care le considerăm semnificative. Într-un calculator electric primitiv, să spunem, informația este indicată de becuri ce se aprind sau se sting după cum curentul trece sau nu. Tensiunea semnalului electric nu contează, și nici sensul curentului. Tot ceea ce contează este că un circuit transmite unul din cele două semnale posibile, din care unul face să se aprindă becul. Distincția pe care o considerăm a fi semnificativă — deosebirea ce permite distingerea, în exprimarea lui Bateson — este cea între prezența sau absența unui curent electric. Definiția lui Bateson este bună, dar formularea sa a însemnat totdeauna ceva mai mult pentru mine. În cursul vieții mele de patru decenii, lumea s-a transformat. Cele mai multe din schimbările pe care le-am văzut în acest răstimp în economie, politică, știință și filozofie au fost cauzate ori au devenit posibile datorită dezvoltării tehnologiei informației. Multe lucruri sînt diferite în lumea de astăzi, dar deosebirea care le-a făcut diferite a fost calculatorul.

În zilele nostre, calculatoarele sînt considerate de mulți ca dispozitive multimedia, capabile de a încorpora și combina toate formele anterioare de comunicare — text, grafică, animație, sunet. Cred că acest punct de vedere conduce la subestimarea potențialului cal-

culatorului. Este adevărat, desigur, că un calculator poate încorpora și manipula toate celelalte mijloace de comunicare, dar adevărata putere a calculatorului constă în capacitatea sa de a manipula nu numai *expresia* ideilor, ci ideile înseși. Pentru mine, uimitor nu este faptul că un calculator poate memora conținutul tuturor cărților dintr-o bibliotecă, ci că el poate observa relațiile dintre conceptele descrise în cărți — nu numai că poate afișa imaginea unei păsări în zbor sau a unei galaxii în rotație, dar el poate concepe și anticipa consecințele legilor fizice care creează aceste adevărate minunății. Calculatorul nu este doar o mașină perfecționată de calculat, de filmat sau de pictat; el este mai degrabă un dispozitiv care accelerează și extinde procesele gîndirii noastre. Este o mașină de conceput imagini mentale, care pornește de la ideile pe care noi le introducem în ea și le duce mai departe decît am fi putut-o face vreodată noi înșine pe cont propriu.

MULȚUMIRI

Acest proiect mi-a fost sugerat inițial de John Brockman, care a simțit nevoia unei cărți scurte rezumînd ideile care se află în spatele calculatoarelor. Mi s-a părut un proiect simplu la acel timp, dar am descoperit curînd că scrierea unei cărți scurte despre un subiect atît de larg poate fi mai dificilă decît scrierea uneia lungi.

Le mulțumesc lui John și William Frucht de la Basic Books pentru ajutorul lor în această strădanie. Am scris cartea în timpul vizitelor la Laboratorul de mijloace de comunicație de la Institutul de Tehnologie din Massachusetts și aș vrea să mulțumesc tuturor profesorilor și studenților care m-au ajutat și sprijinit acolo, îndeosebi directorului și fondatorului laboratorului, Nicholas Negroponte. Am primit un ajutor important în pregătirea manuscrisului inițial de la Debbie Widener și de asemenea de la Bettylou McClanahan și Peggi Oakley. Cea mai mare parte a ceea ce se află în carte am învățat de la prietenul și mentorul meu Marvin Minsky și de la alți profesori de la Institutul de Tehnologie din Massachusetts, care m-au inițiat, inclusiv Gerald Sussman, Claude Shannon, Seymour Papert, Tomaso Poggio, Patrick Winston și Tom Knight.

Aş dori de asemenea să mulţumesc celor ce au citit primele ciorne şi au făcut comentarii folositoare: Jerry Lions, Seymour Papert, George Dyson, Chris Sykes, Brian Eno, Po Bronson, Argye Hillis şi Pati Hillis. Am primit sfaturi utile referitoare la unele capitole de la Tommy Poggio, Neil Gershenfeld, Simon Garfinkel, Mitchell Resnick şi Marvin Minsky. Am fost foarte norocos să primesc ajutor de la Sara Lippincott, care a editat textul, îmbunătăţindu-l mult în cursul acestui proces. În sfîrşit, aş dori să mulţumesc familiei mele: părinţilor mei Argyl şi Bill, care au încurajat interesul meu pentru proiectarea unor maşini complexe; copiilor mei Noah, Asa şi India; şi în special soţiei mele Pati, care m-a încurajat şi sprijinit cu răbdare de-a lungul întregului proiect.

CAPITOLUL 1

Piulițe și șuruburi

Cînd eram copil, am citit o poveste despre un băiat care a construit un robot din piese pe care le-a găsit azvîrlite la vechituri. Robotul băiatului se putea mișca, vorbi și gîndi întocmai ca o persoană, și a devenit prietenul său. Pentru un motiv anume, ideea de a construi un robot mi s-a părut foarte atrăgătoare, așa încît am hotărît să construiesc eu însumi unul. Îmi amintesc cum strîngeam părți ale corpului — tuburi pentru brațe și picioare, motoare pentru mușchi, becuri pentru ochi și o cutie mare de vopsea pentru cap — cu speranța deplină și optimistă că, după ce ele vor fi asamblate și ștecărul introdus în priză, mă voi afla în fața unui om mecanic, care să funcționeze.

După ce era să mă electrocutez de cîteva ori, am început să obțin ca părțile ce le adunasem să se miște, să lumineze și să facă zgomote. Simțeam că progresam. Am început să înțeleg cum să construiesc articulații pentru brațe și picioare. Dar ceva mai important a început să-mi treacă prin minte: nu aveam nici cea mai vagă idee cum să controlez motoarele și luminile și am înțeles că lipsea ceva în cunoștințele mele despre funcționarea roboților. Acum am un nume pentru ceea ce lipsea: *calculul.* Pe atunci îl numeam „gîndire“, și am

înțeles că nu aveam nici un indiciu pentru a face ceva să gîndească. Acum îmi pare evident că partea cea mai dificilă în construirea unui om mecanic este calculul, dar, pentru un copil, aceasta a apărut ca o surpriză.

LOGICĂ BOOLEANĂ

Din fericire, prima carte pe tema calculului pe care am citit-o a fost o carte clasică. Tatăl meu a fost epidemiolog și locuiam în acea vreme la Calcutta. Era dificil de găsit cărți în engleză, dar în biblioteca consulatului britanic am găsit un exemplar prăfuit al unei cărți scrise de logicianul din secolul al nouăsprezecelea George Boole. Titlul cărții a fost cel care m-a atras: *O cercetare a legilor gîndirii*. Acesta mi-a stîrnit imaginația. Puteau într-adevăr exista legi care să determine gîndirea ? În cartea sa, Boole încerca să reducă logica gîndirii umane la operații matematice. Deși nu explica gîndirea umană cu adevărat, Boole demonstra surprinzătoarea putere și generalitate a cîtorva tipuri simple de operații logice. El a inventat un limbaj pentru descrierea și manipularea propozițiilor logice și pentru a determina dacă ele sînt sau nu adevărate. Limbajul este numit acum *algebră booleană*.

Algebra booleană este asemănătoare cu algebra pe care ați învățat-o în școală, cu deosebirea că variabilele din ecuații reprezintă afirmații logice în loc de numere. Variabilele algebrei booleene desemnează propoziții care sînt fie adevărate, fie false, și simbolurile $\wedge$, $\vee$, și $\neg$ reprezintă operațiile logice **Și**, **Sau** și **Nu**. De exemplu, următoarea expresie este o ecuație algebrică booleană

$$\neg (A \vee B) = (\neg A) \wedge (\neg B) .$$

Această ecuație concretă, numită formula lui de Morgan (după Augustus de Morgan, colegul lui Boole), spune că dacă nici *A*, nici *B* nu sînt adevărate, atunci ambele *A* și *B* trebuie să fie false. Variabilele *A* și *B* pot reprezenta orice propoziție logică (adică adevărată sau falsă). Această ecuație concretă este evident adevărată, dar algebra booleană permite de asemenea formularea și demonstrarea sau infirmarea unor propoziții logice cu mult mai complexe.

Opera lui Boole și-a găsit locul în informatică prin teza de specializare a unui tînăr student în inginerie la Institutul de Tehnologie din Massachusetts, pe nume Claude Shannon. Shannon este cel mai bine cunoscut pentru faptul de a fi inventat o ramură a matematicii numită *teoria informației*, care definește măsura pentru informație, pe care o numim *bit*. Inventarea bitului a fost o realizare impresionantă, dar ceea ce a făcut Shannon cu logica booleană a fost cel puțin tot atît de important pentru știința calculului. Cu aceste două realizări Shannon a pus bazele unor progrese care vor apărea în domeniul calculului în următorii aproape cincizeci de ani.

Shannon a fost interesat de construirea unei mașini care să poată juca șah — și, mai general, de construirea unor mecanisme care să imite gîndirea. În 1940 el a publicat lucrarea sa de specializare intitulată „O analiză simbolică a circuitelor cu relee de comutare". În ea a arătat că este posibilă construirea unor circuite electrice echivalente cu expresii din algebra booleană. În circuitele considerate de Shannon, comutatoarele care erau deschise sau închise corespundeau unor variabile logice din algebra booleană, care erau adevărate sau false. Shannon a arătat un mod de a converti orice expresie din algebra booleană într-un aranjament de comu-

tatoare. Circuitul era închis dacă propoziția era adevărată și era deschis dacă propoziția era falsă. Drept consecință a acestei construcții, orice funcție care poate fi descrisă ca o propoziție logică precisă poate fi implementată de un sistem analogic de comutatoare.

În loc de a prezenta detaliat formalismele dezvoltate de Boole și Shannon, voi prezenta un exemplu de aplicare a lor în proiectarea unui model foarte simplu de dispozitiv de calcul, o mașină care joacă „trei-în-linie". Această mașină este mult mai simplă decît un calculator de uz general, dar ea ilustrează două principii care sînt importante pentru orice tip de calculator. Ea arată cum o sarcină poate fi redusă la *funcții logice* și cum astfel de funcții pot fi implementate ca un circuit de comutatoare conectate. De fapt am construit o mașină „trei-în-linie" din becuri și comutatoare la scurt timp după ce am citit cartea lui Boole la Calcutta, iar aceasta a fost intrarea mea în logica calculatorului. Mai tîrziu, cînd eram student în primii ani la Institutul de Tehnologie din Massachusetts, Claude Shannon mi-a devenit profesor și prieten, și am descoperit că și el folosise becuri și comutatoare pentru a construi o mașină care putea juca „trei-în-linie".

După cum cei mai mulți cititori probabil știu, jocul se desfășoară pe o tablă caroiată cu 3 × 3 pătrățele. Jucătorii marchează pe rînd pătrățelele, un jucător folosind o cruciuliță, celălalt un cerculeț. Primul jucător care plasează trei simboluri pe o linie (orizontală, verticală sau pe diagonală) cîștigă partida. Copiii mici se distrează cu „trei-în-linie" deoarece pare a oferi nenumărate strategii posibile de cîștig. În cele din urmă își dau seama că doar un număr mic de poziții pot apărea și prin urmare jocul își pierde farmecul: de îndată ce amîndoi jucătorii învață pozițiile, fiecare partidă se

termină invariabil la egalitate. „Trei-în-linie“ este un exemplu bun de calcul tocmai pentru că el oscilează pe această linie între complex și simplu. Trecerea acelei linii înseamnă tocmai calculul. Calculul înseamnă executarea unor sarcini ce par a fi complexe (precum cîștigarea unei partide de „trei-în-linie“) prin fragmentarea lor în operații simple (ca aceea de a închide un comutator).

În jocul „trei-în-linie“, situațiile care apar sînt suficient de puține pentru a le putea scrie practic pe toate și prin urmare pentru a introduce în mașină răspunsul corect în fiecare caz. Putem folosi un proces simplu în doi pași pentru a proiecta mașina: *în primul rînd* reducem jocul la o serie de cazuri, definind răspunsul corect la fiecare succesiune posibilă de mișcări; *în al doilea rînd* se convertesc acele cazuri în circuite electrice, legînd comutatoarele cu conductori electrici, pentru a recunoaște pozițiile și a indica răspunsul potrivit.

Un mod de a continua ar fi să scriem fiecare aranjament imaginabil de cruciulițe și cerculețe care ar putea fi plasat pe tablă și apoi să decidem cum ar putea juca în fiecare situație calculatorul. Deoarece fiecare din cele nouă pătrățele are trei stări posibile (cruciuliță, cerculeț sau liber), sînt 3^9 (sau 19 683) moduri de a completa tabla. Dar cele mai multe dintre aceste poziții nu ar apărea niciodată în cursul unei partide. O metodă mai bună de listare a posibilităților este să trasăm un *arbore al partidei* — o configurație care urmărește fiecare linie posibilă de joc. Arborele partidei are ca rădăcină o tablă liberă și are o ramură pentru fiecare linie alternativă de joc determinată de mutările jucătorului uman. (Arborele nu trebuie să se ramifice cînd joacă mașina deoarece răspunsul mașinii la orice mișcare dată este totdeauna determinat dinainte.) Figura 1 arată

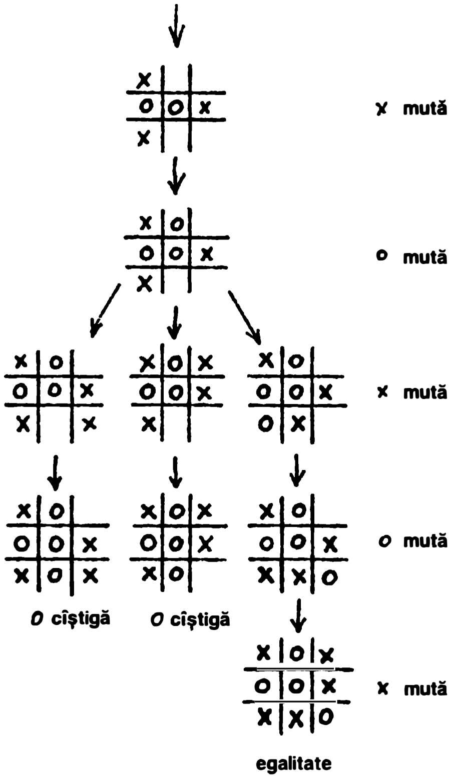


FIGURA 1 Parte a unui arbore al partidei de „trei-în-linie"

o mică parte a unui astfel de arbore. Pentru fiecare mutare posibilă făcută de cruciuliță, jucătorul uman, există un răspuns determinat dinainte pentru maşină. (Dintr-un motiv ciudat, informaticienii desenează arborii totdeauna de sus în jos, cu „rădăcina" în sus.)

Arborele din Figura 1 ilustrează strategia pe care o folosesc întotdeauna la „trei-în-linie": joc în centru de cîte ori pot. Mişcările maşinii sînt determinate de mişcările jucătorului uman, ceea ce reduce enorm numă-

rul posibilităților de avut în vedere. Un arbore complet al partidei, arătînd ce ar trebui să facă mașina în fiecare situație, are cam cinci sute sau șase sute de ramuri, numărul exact depinzînd de amănuntele strategiei. Urmărirea arborelui va conduce mașina la a cîștiga, sau cel puțin la a nu pierde, fiecare partidă. Regulile jocului sînt încorporate în răspunsuri, așa că, urmărind arborele, mașina va respecta regulile întotdeauna. Din acest arbore al partidei putem formula descrieri detaliate a ceea ce ar trebui să joace mașina în fiecare poziție concretă. Aceste descrieri detaliate constituie logica booleană a mașinii.

De îndată ce am definit comportamentul dorit, putem traduce acel comportament în circuite electrice construite din baterii, fire, comutatoare și becuri. În mașină, circuitul fundamental este același circuit folosit într-o lanternă: cînd comutatorul este apăsat — adică închis — lumina se aprinde deoarece un circuit complet s-a format între bec și baterie. (Legăturile la baterie sînt indicate cu semnele + și –.) Cel mai important, aceste comutatoare pot fi conectate fie în serie, fie în paralel. De exemplu, putem pune în serie două comutatoare pentru a obține lumină doar cînd ambele comutatoare sînt închise. Acest circuit implementează una din funcțiile de comutare fundamentale ale calculatorului — „blocul logic“ cunoscut ca funcția **Și**, numit astfel deoarece becul luminează numai cînd primul **și** al doilea comutator sînt închise. Comutatoarele conectate în paralel implementează funcția **Sau**, care închide circuitul (și astfel becul luminează) de cîte ori unul **sau** amîndouă comutatoarele sînt închise (*cf.* Figura 2).

Aceste modele simple de conectare în serie sau în paralel pot fi folosite în combinații pentru a forma

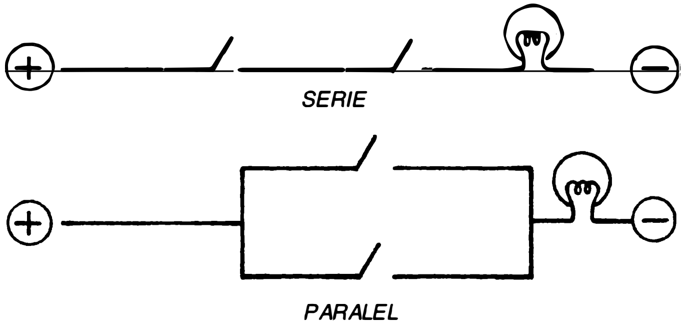


FIGURA 2 Comutatoare în serie și în paralel

conexiuni care acționează conform cu diverse reguli logice. În mașina „trei-în-linie", lanțuri de comutatoare legate în serie sînt folosite pentru a detecta poziții, și aceste lanțuri sînt conectate în paralel la becuri, astfel încît mai multe poziții pot aprinde același bec — adică produc același răspuns din partea mașinii.

Mașina „trei-în-linie" pe care am construit-o are patru grupuri cu cîte nouă comutatoare fiecare, și fiecare comutator corespunde unuia din cele nouă pătrățele ale tablei de „trei-în-linie". Ea are de asemenea nouă becuri aranjate sub forma unei table de „trei-în-linie". Mașina, care întotdeauna joacă prima, indică mutările sale aprinzînd un bec. Jucătorul uman mută închizînd un comutator — folosind primul grup de comutatoare pentru prima sa mutare, al doilea grup pentru a doua sa mutare, și așa mai departe. În versiunea mea, mașina începe întotdeauna jucînd în colțul din stînga sus al tablei, o schemă care micșorează considerabil numărul de cazuri. Jucătorul uman răspunde închizînd unul din comutatoarele din primul grup (să spunem, cel corespunzător pătrățelului din centrul tablei), și partida continuă. Strategia mașinii este încorporată în legăturile dintre comutatoare și becuri.

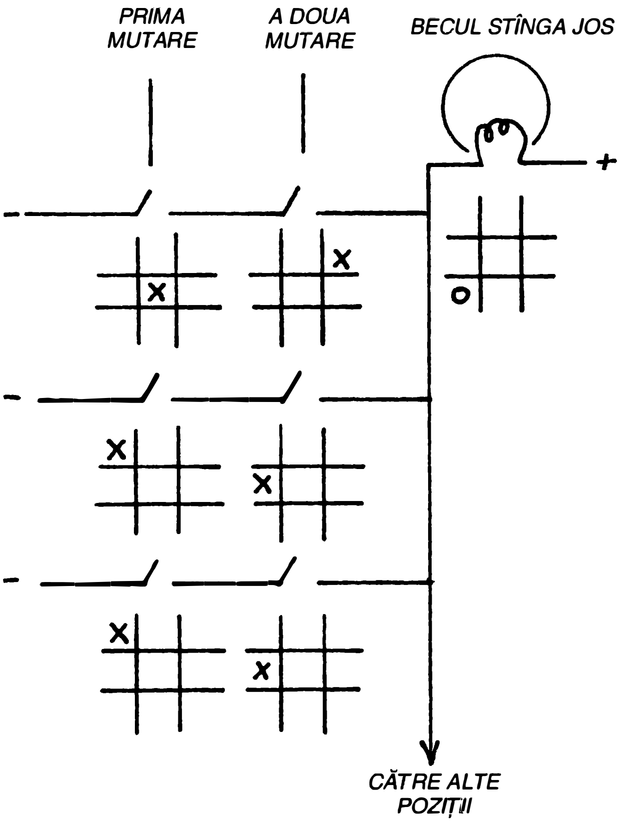


FIGURA 3 Mai multe poziții diferite care produc același răspuns

Legătura care produce primul răspuns al mașinii este simplă (*cf.* Figura 3). Fiecare comutator din primul grup este legat la un bec care corespunde răspunsului mașinii. De exemplu, o mutare în centru provoacă un răspuns în pătrățelul din dreapta jos, astfel încît comutatorul central este legat la becul din dreapta jos. Deoarece mașina mea răspunde întotdeauna în pătrățelul central dacă poate, cele mai multe comutatoare

din primul grup sînt legate în paralel cu becul din mijloc.

Fiecare poziție pentru a doua rundă a partidei depinde de primele două mutări ale jucătorului uman. Pentru a recunoaște această combinație de mutări ale jucătorului uman, comutatoarele corespunzătoare sînt legate în serie. De exemplu, dacă prima mutare a jucătorului este în centru și a doua în dreapta sus, atunci mașina trebuie să răspundă jucînd în stînga jos. Această poziție este realizată legînd în serie comutatorul central din primul grup cu comutatorul dreapta sus din al doilea grup („dacă pătrățelele din centru **și** dreapta sus sînt ocupate, atunci ...“), cu lanțul de două comutatoare legat la becul din stînga jos. Fiecare legătură în paralel la un bec precizează o combinație diferită care va produce aprinderea becului („această mutare sau acea mutare vor provoca acest răspuns“). De cîte ori a fost necesar să folosesc același comutator în două circuite diferite, am utilizat un comutator cu dublă ieșire — două comutatoare legate mecanic la același buton, astfel că sînt acționate împreună — ceea ce permite ca aceeași mutare să facă parte din două poziții diferite. Legăturile celui de-al treilea și ale celui de-al patrulea grup de comutatoare respectă același principiu, dar sînt chiar mai multe combinații. Cum vă puteți imagina, legarea devine complicată, chiar dacă principiile sînt simple. Pe tablă sînt mai puține opțiuni posibile, dar lanțurile de comutatoare sînt mai lungi.

Mașina „trei-în-linie“ pe care am construit-o are cam o sută cincizeci de comutatoare. La acea vreme mi se părea mult (am făcut comutatoarele din lemn și cuie), dar cipurile de calculator pe care le proiectez astăzi au milioane de comutatoare, cele mai multe dintre ele conectate în modele foarte asemănătoare cu cele folo-

site în mașina „trei-în-linie". Cele mai multe calculatoare moderne folosesc un model diferit de comutator electric — un tranzistor, pe care îl voi descrie mai tîrziu — dar ideea fundamentală este exact aceeași: a lega comutatoarele în serie pentru a produce o funcție **Și** și a lega comutatoarele în paralel pentru a produce o funcție **Sau**.

Deși logica mașinii „trei-în-linie" este asemănătoare cu logica unui calculator, există mai multe diferențe importante. Una este aceea că mașina „trei-în-linie" nu are ideea de evenimente petrecîndu-se secvențial în timp; de aceea, întreaga secvență a partidei — adică întregul arbore al partidei — trebuie determinată dinainte. Acest lucru este destul de incomod cînd e vorba de „trei-în-linie" și practic imposibil pentru jocurile mai complicate, cum ar fi șah sau chiar dame. Calculatoarele moderne sînt foarte bune la jocul de dame și destul de bune la șah (*cf.* capitolul 5), deoarece, în loc de un arbore de joc predeterminat, ele folosesc o metodă diferită — una care include examinarea pozițiilor secvențial în timp.

O altă diferență între mașina „trei-în-linie" și un calculator de uz general este că mașina „trei-în-linie" poate executa numai o funcție. „Programul" mașinii este încorporat în circuitele sale. Mașina „trei-în-linie" nu are software.

BIȚI ȘI BLOCURI LOGICE

Cum am arătat în introducere, nu există nici un motiv pentru care mașina „trei-în-linie" (sau orice alt calculator) trebuie să fie construită din comutatoare electrice. Un calculator poate reprezenta informații folosind

curenți electrici, presiunile din fluide sau chiar reacții chimice. Indiferent dacă dumneavoastră construiți un calculator din tranzistori, valve hidraulice sau o trusă chimică, principiile după care el funcționează sînt în mare parte aceleași. Ideea fundamentală a mașinii „trei-în-linie“ este aceea că funcția **Și** este implementată legînd în serie două comutatoare și că funcția **Sau** este implementată legînd în paralel două comutatoare, dar există multe alte moduri de a implementa funcțiile **Și** și **Sau**.

Aici trebuie să fac o paranteză pentru a menționa *bitul*. Cea mai mică „deosebire ce permite distingerea“ (pentru a folosi din nou fraza lui Bateson) este diferența care împarte toate semnalele în două clase distincte. În mașina „trei-în-linie“, cele două clase sînt „curentul circulă“ și „curentul nu circulă“. Prin convenție, numim cele două clase posibile 1 și 0. Acestea sînt doar nume; am putea la fel de bine să le denumim **Adevărat** și **Fals**, sau **Alice** și **Bob.** Însăși alegerea clasei pe care o numim 0 și a celei pe care o numim 1 este arbitrară. Un semnal care poate purta unul din cele două mesaje diferite (cum ar fi 1 sau 0) este numit semnal *binar* sau *bit*. Un calculator folosește combinații de biți pentru a reprezenta orice fel de mulțimi de alternative — diferite mutări la „trei-în-linie“, să spunem, sau diferite culori de afișat pe ecran. Deoarece convenția este de a desemna biții prin unuri și zerouri, oamenii consideră adeseori aceste secvențe de biți drept numere, de unde vechea vorbă „Calculatorul face totul cu numere“. Dar această convenție este numai un mod de a gîndi asupra a ceea ce se petrece. Dacă am fi numit X și Y cele două mesaje posibile transmise de bit, oamenii ar fi spus

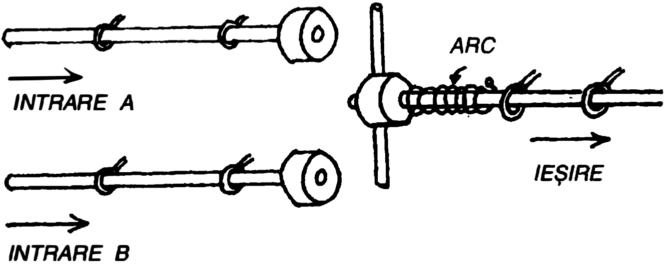


FIGURA 4 Implementarea mecanică a funcției **Sau**

„Calculatorul face totul cu litere". Afirmația cea mai exactă este „Calculatorul reprezintă numerele, literele și orice altceva prin secvențe de biți".

În loc de a folosi curentul electric pentru a reprezenta un bit, am fi putut folosi mișcarea mecanică. Figura 4 arată cum este implementată funcția **Sau** folosind o tehnologie care reprezintă 1 prin alunecarea spre dreapta a unei tije. Cîtă vreme ambele tije de intrare *A* și *B* se află la stînga, reprezentînd 0, arcul va ține tija de ieșire împinsă la stînga, dar dacă oricare din tijele de intrare alunecă spre dreapta, atunci tija de ieșire va aluneca de asemenea spre dreapta. Obiectul din Figura 5 calculează altă funcție utilă, cea de inversare: inversorul transformă fiecare semnal în contrariul său; de exemplu, el transformă apăsatul spre dreapta în tras spre stînga și viceversa.

Aceste funcții **Și**, **Sau** și **Invers** sînt *blocuri logice*, și ele pot fi conectate pentru a crea alte funcții. De exemplu, ieșirea unui bloc **Sau** poate fi legată la un bloc **Invers** pentru a crea o funcție **Nor**: ieșirea funcției **Nor** va fi 1 cînd nici una din intrări nu este 1. În alt exemplu (folosind formula lui de Morgan), putem realiza un bloc **Și** prin legarea a două blocuri **Invers** la intrările unui bloc **Sau** și prin conectarea unui al

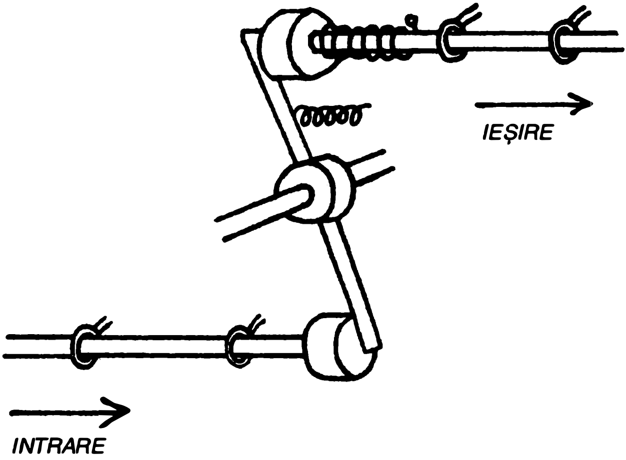


FIGURA 5 Inversor mecanic

treilea bloc **Invers** la ieșire (*cf.* Figura 6). Aceste patru blocuri funcționează împreună pentru a implementa funcția **Și**, astfel că ieșirea finală este 1 numai cînd ambele intrări sînt 1.

Primele dispozitive de calcul au fost realizate din componente mecanice. În secolul al șaptesprezecelea Blaise Pascal a construit o mașină mecanică de adunat care i-a inspirat pe Gottfried Wilhelm Leibniz și pe matematicianul englez Robert Hooke să construiască mașini îmbunătățite care puteau înmulți, împărți și chiar extrage rădăcina pătrată. Aceste mașini nu erau programabile, dar în 1833, un alt englez, matematicianul și inventatorul Charles Babbage, a proiectat și în parte a construit un calculator mecanic programabil. Chiar și pe vremea copilăriei mele, în anii șaizeci, cele mai multe calculatoare aritmetice erau mecanice. Întotdeauna mi-au plăcut aceste mașini mecanice deoarece poți vedea ce se petrece, ceea ce nu este cazul cu calcu-

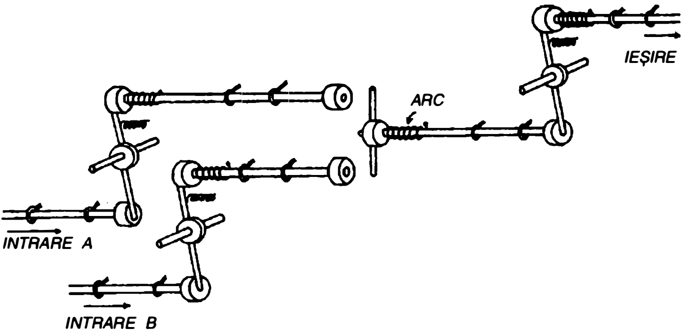


FIGURA 6 Un bloc **Și** construit legînd un bloc **Sau** la inversori

latoarele electronice. Cînd proiectez un cip pentru un calculator electronic, îmi imaginez acțiunea circuitelor ca părți mecanice în mișcare.

CALCULATORUL HIDRAULIC

Imaginea pe care o am în minte cînd proiectez un circuit logic este cea a valvelor hidraulice. O valvă hidraulică este asemănătoare cu un comutator care controlează și este controlat de curentul de apă. Fiecare valvă are trei legături: intrarea, ieșirea și controlul. Presiunea de pe legătura de control împinge un piston care închide fluxul de apă de la intrare la ieșire. Figura 7 arată un circuit pentru funcția **Sau** construit din valve hidraulice.

În acest circuit, presiunea apei este folosită pentru a distinge cele două semnale posibile. Observați că, într-o valvă hidraulică, conducta de control poate influența conducta de ieșire, dar conducta de ieșire nu poate influența conducta de control. Această restricție stabilește un flux ascendent de informație prin comu-

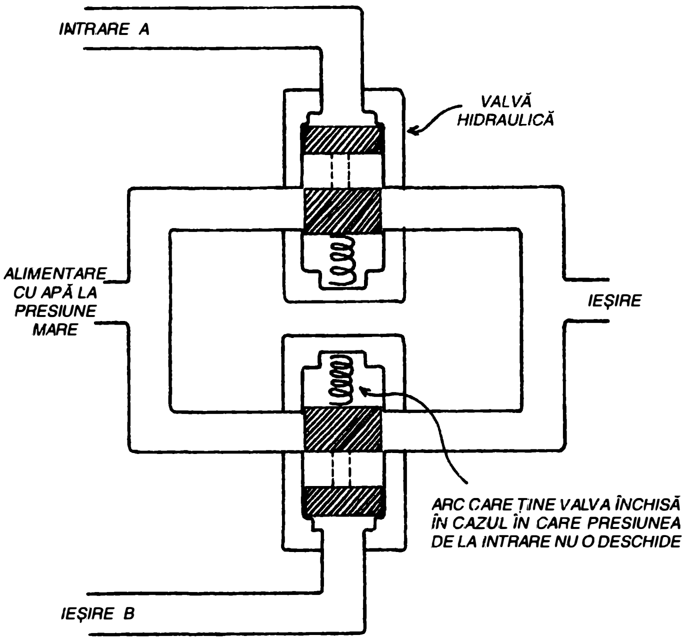


FIGURA 7 Un bloc **Sau** construit cu valve hidraulice

tator; într-un sens, ea stabilește o direcție în timp. Întrucît valva este fie deschisă, fie închisă, ea îndeplinește de asemenea o funcție suplimentară de *amplificare*, ce permite ca puterea semnalului să fie restabilită la valoarea sa maximă la fiecare stadiu. Chiar dacă intrarea are o presiune mică — deoarece trece printr-o conductă lungă și subțire, să spunem, sau datorită unei scurgeri — ieșirea va fi totdeauna la presiunea completă datorită acțiunii pornit-oprit a valvei. Aceasta este diferența fundamentală între *numeric* și *analogic*: o valvă numerică este fie deschisă, fie închisă; o valvă analogică, asemeni robinetului din bucătăria dumneavoastră, poate fi în orice poziție intermediară. În cal-

culatorul hidraulic, tot ceea ce i se cere semnalului de intrare este să fie suficient de puternic pentru a mișca valva. În acest caz, deosebirea care permite distingerea este diferența în presiunea apei, suficientă pentru a deschide valva. Și întrucît un semnal de intrare slăbit va produce, totuși, o ieșire cu puterea completă, putem lega mii de straturi de logică, ieșirea unui strat controlînd următorul strat, fără a fi preocupați de descreșterea treptată a presiunii. Ieșirea fiecărei porți va fi întotdeauna la presiunea completă.

Acest tip de proiect este numit *logică cu restabilire* și exemplul din tehnologia hidraulică este deosebit de interesant deoarece corespunde aproape exact logicii folosite în calculatoarele electronice moderne. Presiunea apei în conducte este analoagă tensiunii din firele electrice și valva hidraulică este analoagă tranzistorului din oxid de metal. Legăturile de control, de intrare și de ieșire ale valvei corespund îndeaproape celor trei legături (numite *emitor*, *bază* și *colector*) ale unui tranzistor. Analogia între valvele hidraulice și tranzistori este atît de exactă încît ați putea traduce proiectul unui microprocesor modern direct în proiectul unui calculator hidraulic. Pentru a face aceasta, ar trebui să priviți la microscop modelul de fire de pe o pastilă de siliciu și apoi să îndoiți o mulțime de conducte în aceleași forme ca și ale sîrmelor de pe pastilă și să le legați exact în același mod. În locul fiecărui tranzistor ați folosi o valvă hidraulică. Conducta care corespunde tensiunii de alimentare a cipului dumneavoastră ar fi legată cu o sursă de alimentare cu apă sub presiune și conducta corespunzătoare legăturii de împămîntare ar putea fi un tub de scurgere.

Pentru a folosi un calculator hidraulic ar trebui să legați echivalentele hidraulice ale dispozitivelor de intrare

şi ieşire — ar trebui să construiţi o tastatură hidraulică, un ecran hidraulic, cipuri de memorie hidraulice, şi aşa mai departe — dar dacă aţi făcut toate acestea, el ar trece prin exact aceleaşi evenimente de comutare ca şi microprocesorul electronic. Desigur, calculatorul hidraulic ar fi mult mai lent decît microprocesorul dumneavoastră cel mai recent (să nu pomenim că ar fi mai mare), deoarece presiunea apei se propagă de-a lungul conductelor mult mai încet decît circulă electricitatea prin fire. În ceea ce priveşte mărimea: întrucît un microcip modern are mai multe milioane de tranzistori, echivalentul său hidraulic ar necesita mai multe milioane de valve. Un tranzistor dintr-un cip are dimensiunea de aproximativ o milionime de metru; o valvă hidraulică are cam zece centimetri pe o latură. Dacă se măresc proporţional conductele, atunci un calculator hidraulic ar acoperi cam un kilometru pătrat cu conducte şi valve. Privit din avion, el ar arăta aproximativ la fel cu un cip electronic privit la microscop.

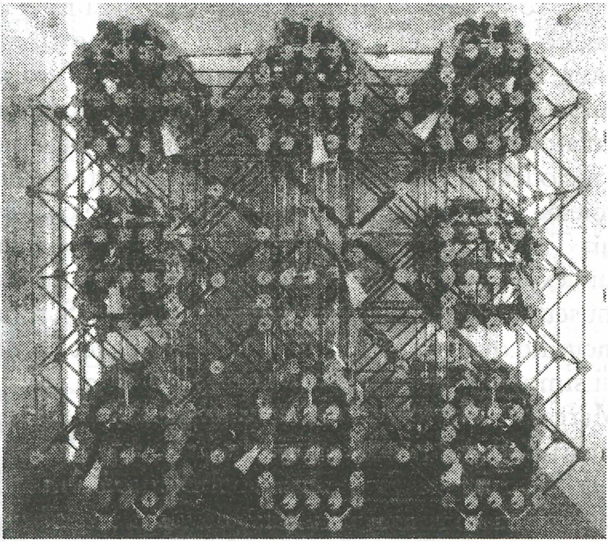
Cînd proiectez un cip de calculator, trasez linii pe un ecran de calculator şi modelul este redus (printr-un proces analog cu mărirea fotografică) şi gravat pe o pastilă de siliciu. Pentru mine, liniile de pe ecran sînt conductele şi valvele. De fapt, cei mai mulţi proiectanţi de calculatoare nici nu se mai obosesc să deseneze linii; în schimb ei descriu exact legăturile între blocurile **Şi** şi **Sau** şi lasă unui calculator sarcina de a elabora detaliile amplasării şi geometriei comutatoarelor. În cea mai mare parte a timpului ei ignoră tehnologia şi se concentrează asupra funcţionalităţii. Şi eu procedez astfel uneori, dar încă prefer să trasez propriile mele forme. De cîte ori proiectez un cip, primul lucru pe care vreau să-l fac este să îl privesc la un microscop — nu pentru

că aș crede că pot învăța ceva nou privindu-l, ci pentru că întotdeauna sînt fascinat de felul în care un model poate crea realitatea.

TINKER TOYS

Cu excepția miniaturizării surprinzător de bune, nu există un motiv special pentru a construi calculatoarele cu tehnologia siliciului. Construirea unui calculator cu orice tehnologie necesită o cantitate mare din două feluri de elemente: *comutatoare* și *conectoare*. Comutatorul este un element de ghidare (valva hidraulică sau tranzistorul) care poate combina semnale multiple într-un singur semnal. Un comutator ideal ar trebui să fie asimetric, astfel încît semnalul de intrare să

Figura 8 Calculator Tinker Toy



influenţeze semnalul de ieşire, dar nu viceversa, şi ar trebui să aibă calitatea de restabilire, astfel că un semnal de intrare slab sau degradat nu va avea drept rezultat o ieşire degradată. Cel de-al doilea element, conectorul, este firul sau conducta care transportă un semnal între comutatoare. Acest element de conectare trebuie să aibă posibilitatea de ramificare, astfel încît o singură ieşire să poată alimenta multe intrări. Cele două sînt singurele elemente necesare pentru a construi un calculator. Mai tîrziu vom prezenta încă un element — un registru pentru stocarea informaţiei — dar acesta poate fi construit din aceleaşi componente de ghidare şi conectare.

Nu am construit niciodată un calculator hidraulic, dar o dată, cu nişte prieteni, am construit un calculator din tije şi fire. Piesele proveneau dintr-un set de construcţii pentru copii numit Tinker Toys. Poate că cititorii îşi amintesc acest set de tije cilindrice de lemn care se fixează în nişte butuci groşi şi mici din lemn, cu găuri în ei. Logica pentru calculatorul meu Tinker Toy funcţiona foarte asemănător cu cea din Figura 8. Ca şi un calculator cu comutatoare şi becuri, calculatorul Tinker Toy juca „trei-în-linie". Nu a pierdut niciodată. A fost foarte dificil să fac calculatorul, fiind nevoie de zeci de mii de piese din mai mult de o sută de seturi Tinker Toy „Giant Engineer" şi produsul finit (acum expus la Muzeul calculatorului din Boston, Massachusetts) arată complicat, pare de neînţeles. Şi totuşi, principiile după care funcţionează sînt exact combinaţii simple de funcţii **Şi** şi **Sau** descrise mai sus.

Marea greşeală pe care am făcut-o la proiectarea calculatorului Tinker Toy este că nu am folosit *logica cu restabilire* — adică nu exista amplificare de la un stadiu al logicii la următorul. Implementarea logicii s-a bazat

pe tije presînd alte tije într-un proiect asemănător cu cel înfățișat în Figura 4. Din cauza acestei alegeri făcute la proiectare, întreaga forță necesară pentru a mișca sutele de elemente ale mașinii trebuia furnizată prin apăsarea comutatorului de intrare. Forța acumulată tindea să întindă firele care transmiteau mișcarea, și, pentru că nu avea loc restabilirea la fiecare stadiu, erorile provocate de întindere se acumulau de la un element logic la următorul. Dacă nu s-ar regla constant firele, mașina ar face greșeli.

Am construit ulterior o versiune a calculatorului Tinker Toy care înlătura neajunsul, dar nu am uitat niciodată lecția acelei prime mașini: tehnologia de implementare trebuie să producă ieșiri perfecte din intrări imperfecte, distrugînd în fașă micile erori. Aceasta este esența tehnologiei numerice, care restabilește la fiecare stadiu semnale aproape de perfecțiune. Este singurul mod pe care îl cunoaștem — cel puțin pînă acum — pentru a ține sub control un sistem complicat.

LIBERI SĂ NE PREOCUPE DEOSEBIREA CE PERMITE DISTINGEREA

Numirea celor două semnale din logica calculatorului 0 și 1 este un exemplu de abstractizare funcțională. Ea ne permite să manipulăm informația fără a fi preocupați de amănuntele reprezentării sale, pe care se bazează. De îndată ce ne imaginăm cum să realizăm o funcție dată, putem pune mecanismul într-o „cutie neagră“ sau într-un „bloc constructiv“ și să încetăm a ne mai gîndi la el. Funcția întruchipată de blocul constructiv poate fi folosită în repetate rînduri fără să se țină seama de detaliile a ceea ce se află înăuntru. Acest

proces de abstractizare funcțională este fundamental în proiectarea calculatoarelor — nu singurul mod de a proiecta sisteme complicate, dar cel mai obișnuit (mai tîrziu voi descrie o metodă alternativă). Calculatoarele sînt constituite dintr-o ierarhie de astfel de abstractizări funcționale, fiecare întruchipată într-un bloc constructiv. Blocurile care execută funcții sînt prinse împreună pentru a implementa funcții mai complexe, și aceste colecții de blocuri devin la rîndul lor noile blocuri constructive pentru nivelul următor.

Această structură ierarhică de abstractizări este instrumentul nostru cel mai puternic în înțelegerea sistemelor complexe, deoarece ea permite să ne concentrăm la un moment dat asupra unui singur aspect al problemei. De exemplu, putem vorbi despre funcții booleene ca **Și** și **Sau** la modul abstract, fără a fi preocupați dacă ele sînt construite din comutatoare electrice, sau bastoane și fire, sau valve acționate hidraulic. În cele mai multe situații putem uita de tehnologie. Acest lucru este minunat, deoarece înseamnă că aproape tot ceea ce spunem despre calculatoare va fi adevărat chiar și atunci cînd tranzistorii și pastilele de siliciu devin perimate.

CAPITOLUL 2

Blocuri constructive universale

De acum încolo putem ignora firele și comutatoarele și putem lucra cu ideea abstractă de blocuri logice operînd cu zerouri și unuri, un pas simplu care ne permite să trecem din domeniul ingineriei în domeniul matematicii. Acesta este capitolul cel mai abstract al cărții; el vă va arăta cum metodele folosite pentru a construi o mașină de jucat „trei-în-linie" pot fi folosite pentru a construi aproape orice funcție. În acest capitol vom defini o mulțime puternică de blocuri constructive: funcțiile logice și automatele cu un număr finit de stări. Cu aceste elemente este ușor de construit un calculator.

FUNCȚII LOGICE

La construirea mașinii de jucat „trei-în-linie" am început prin a scrie arborele partidei, care ne-a dat o mulțime de reguli pentru a produce ieșirile din intrări. Această metodă de abordare se dovedește a fi folositoare în general. De îndată ce scriem regulile care precizează ce ieșiri dorim pentru fiecare combinație a intrărilor, putem construi un dispozitiv ce implementează aceste reguli folosind funcțiile **Și, Sau** și **Invers**. Blocurile logice **Și, Sau** și **Invers** formează o *mulțime con-*

structivă universală, care poate fi folosită pentru a implementa orice mulțime de reguli. (Aceste tipuri simple de blocuri logice sînt numite uneori și *porți logice*.)

Ideea unei mulțimi universale de blocuri este importantă: înseamnă că mulțimea este suficient de generală pentru a construi orice. Cînd eram copil, jucăria mea preferată era un set de piese din plastic cu prindere reciprocă numite blocuri Lego, cu care am construit tot felul de jucării: mașini, case, nave cosmice, dinozauri. Îmi plăcea să mă joc cu aceste blocuri, dar ele nu erau întru totul universale, deoarece singurele obiecte pe care le puteai construi cu ele aveau un aspect colțuros, în trepte. Construirea a ceva cu o formă diferită — un cilindru sau o sferă, de exemplu — ar fi necesitat un tip nou de blocuri. În cele din urmă a trebuit să trec la alt mediu pentru a construi lucrurile pe care le doream. Dar blocurile **Și, Sau** și **Invers** din logica booleană sînt o mulțime constructivă universală pentru a transforma intrările în ieșiri. Pentru a vedea cum formează ele o mulțime universală, cel mai bine este să înțelegem o metodă generală de a le folosi la implementarea regulilor. Pentru început vom considera reguli *binare* — reguli care specifică intrări și ieșiri care sînt fie 0, fie 1. Mașina de jucat „trei-în-linie" este un bun exemplu de funcție specificată de reguli binare, deoarece comutatoarele de la intrare și becurile de la ieșire sînt fie deschise, fie închise — adică fie 1, fie 0. (Mai tîrziu vom discuta regulile pentru a manipula litere, cifre, sau chiar imagini și sunete, ca intrări și ieșiri.) Orice mulțime de reguli binare poate fi complet specificată prin prezentarea unui tabel cu ieșirile pentru fiecare combinație posibilă de zerouri și unuri la intrare. De exemplu, regulile pentru funcția **Sau** sînt descrise detaliat în următorul tabel:

	Intrare A	Intrare B	Ieșire
	0	0	0
Funcția Sau	0	1	1
	1	0	1
	1	1	1

Funcția **Invers** este specificată de un tabel chiar mai simplu:

	Intrare	Ieșire
Funcția Invers	0	1
	1	0

Pentru o funcție binară cu *n* intrări sînt posibile 2^n combinații ale semnalelor de intrare. Uneori nu va avea rost să le precizăm pe toate, întrucît anumite combinații ale intrărilor nu ne interesează. De exemplu, la specificarea funcției realizate de către mașina de jucat „trei-în-linie" nu ne interesează ce se întîmplă dacă jucătorul uman joacă în toate pătrățelele simultan. Această mutare nu este admisă și nu trebuie să precizăm ieșirea funcției pentru această combinație a intrărilor.

Blocurile logice complexe sînt construite conectînd blocuri **Și, Sau** și **Invers.** În desenele pentru modelul conexiunii, cele trei blocuri sînt reprezentate convențional prin cutii de forme diferite (*cf.* Figura 9);

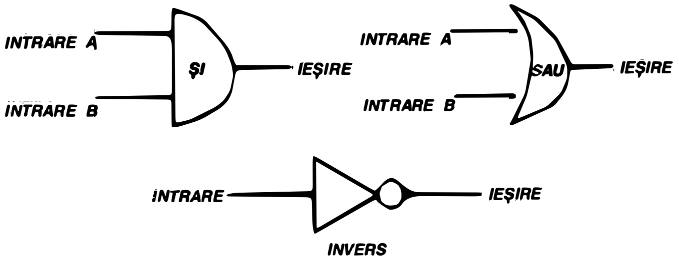


FIGURA 9 Blocuri **Și, Sau** și **Invers**

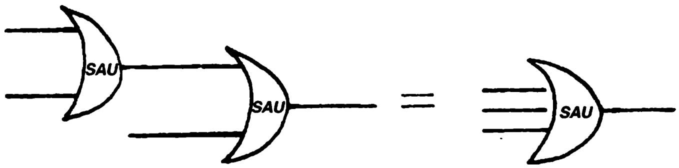


Figura 10 Un bloc **Sau** cu trei intrări obținut dintr-o pereche de blocuri **Sau** cu două intrări

segmentele de dreaptă legate în partea stîngă reprezintă intrări în blocuri, iar segmentele legate în partea dreaptă reprezintă ieșirea. Figura 10 arată cum pot fi conectate o pereche de blocuri **Sau** cu două intrări pentru a forma o funcție **Sau** cu trei intrări; ieșirea pentru această funcție va fi 1 dacă *oricare* din cele trei intrări este 1. De asemenea, este posibil a conecta în mod asemănător cîteva blocuri **Și** pentru a obține un bloc **Și** cu orice număr de intrări.

Figura 11 arată cum poate fi construit un bloc **Și** prin conectarea unui bloc **Invers** la fiecare din intrările și la ieșirea unui bloc **Sau**. (Aici apare încă o dată formula lui de Morgan.) Pentru a înțelege cum funcționează acesta, cel mai bine este să urmărim cifrele zero și unu pentru fiecare combinație de intrări. Observați că această ilustrație este practic aceeași cu Figura 6 din capitolul precedent. Acest lucru atrage atenția asupra unui fapt interesant: nu avem neapărată nevoie de blocuri

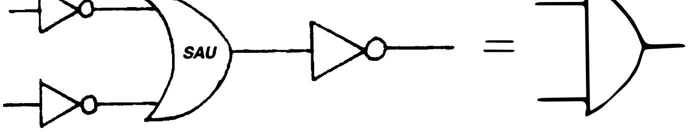


Figura 11 Transformarea lui **Sau** în **Și**

Și în mulțimea noastră constructivă universală deoarece le putem construi totdeauna din blocuri **Sau** și **Invers**.

Ca și la mașina de jucat „trei-în-linie", blocurile **Și** sînt folosite pentru a detecta toate combinațiile posibile ale intrărilor pentru care ieșirea este 1, în vreme ce blocurile **Sau** furnizează o listă cu aceste combinații. Să pornim, de exemplu, cu o funcție simplă cu trei intrări. Să ne imaginăm că dorim să construim un bloc ce permite celor trei intrări să voteze ieșirea. În acest bloc nou, majoritatea cîștigă — adică ieșirea va fi 1 numai dacă două sau mai multe dintre intrări sînt 1.

Intrări A B C	**Majoritate Ieșiri**
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

Figura 12 A arată cum este implementată această funcție. Un bloc **Și** cu blocuri **Invers** adecvate la intrare este folosit pentru a recunoaște fiecare combinație de intrări pentru care ieșirea este 1; aceste blocuri sînt conectate printr-un bloc **Sau** care produce ieșirea. Această strategie poate fi folosită pentru a crea orice transformare a intrărilor în ieșiri.

Desigur, această metodă concretă de folosire a porților **Și** separate pentru a recunoaște fiecare combinație a intrărilor nu este singurul mod de implementare a funcției și adesea nu este nici cel mai simplu mod. Figura 12 B arată un mod mai simplu de a produce funcția

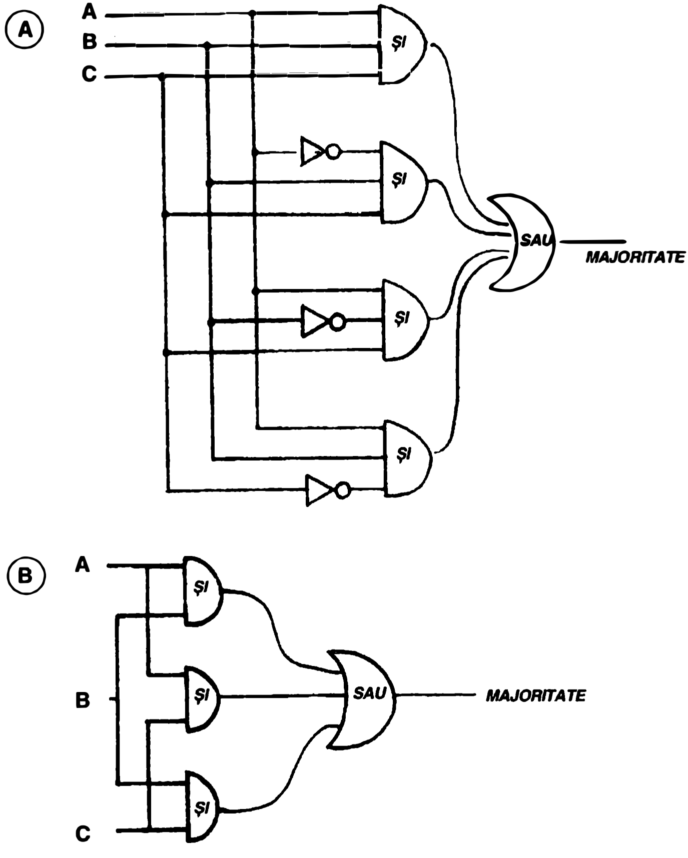


FIGURA 12 Implementarea funcției majoritate cu blocuri **Și**, **Sau** și **Negație**

majoritate. Cel mai important lucru referitor la metoda descrisă nu este faptul că produce implementarea cea mai bună, ci că produce întotdeauna o implementare ce funcționează. Concluzia importantă ce trebuie trasă este aceea că e posibilă combinarea blocurilor **Și**, **Sau** și **Invers** pentru a realiza *orice* funcție binară — adică

orice funcție care poate fi precizată printr-o tabelă de intrări-ieșiri cu zerouri și unuri.

Limitarea intrărilor și ieșirilor la numere binare nu este în realitate o restricție puternică, deoarece combinațiile de zerouri și unuri pot fi folosite pentru a reprezenta alte lucruri — litere, numere mai mari, orice entitate care poate fi codificată. Ca exemplu de funcție nebinară, să presupunem că dorim să construim o mașină care acționează ca un judecător în jocul copiilor „foarfeca — hîrtia — piatra". Acesta este un joc pentru două persoane, în care fiecare jucător alege în secret una din cele trei „arme" — foarfecă, hîrtie sau piatră. Regulile sînt simple: foarfeca taie hîrtia, hîrtia acoperă piatra, piatra zdrobește foarfeca. Dacă cei doi copii aleg aceeași armă, ei sînt la egalitate. În loc să construim o mașină care joacă acest joc (ceea ce ar implica ghicirea armei pe care o va alege adversarul), vom construi o mașină care stabilește cine cîștigă. Aici este tabelul cu intrările și ieșirile pentru funcția care are ca intrări armele alese și care declară învingătorul ca ieșire. Tabela codifică regulile jocului:

Intrare A	**Intrare B**	**Ieșire**
Foarfecă	Foarfecă	Egalitate
Foarfecă	Hîrtie	Cîștigă A
Foarfecă	Piatră	Cîștigă B
Hîrtie	Foarfecă	Cîștigă B
Hîrtie	Hîrtie	Egalitate
Hîrtie	Piatră	Cîștigă A
Piatră	Foarfecă	Cîștigă A
Piatră	Hîrtie	Cîștigă B
Piatră	Piatră	Egalitate

Funcția judecător pentru jocul „foarfecă — hîrtie — piatră" este o funcție combinatorică, dar nu este o funcție binară, deoarece intrările și ieșirile sale au mai mult de

două valori posibile. Pentru a implementa această funcție ca un bloc logic combinatoric, trebuie să o transformăm într-o funcție de zerouri și unuri. Pentru aceasta, trebuie să stabilim niște convenții pentru reprezentarea intrărilor și ieșirilor. Un mod simplu de a face acest lucru ar fi să folosim un bit separat pentru fiecare dintre posibilități. Ar fi trei semnale de intrare pentru fiecare armă: un 1 la prima intrare reprezintă **Foarfeca**, un 1 la a doua intrare reprezintă **Piatra** și un 1 la a treia intrare reprezintă **Hîrtia**. Am putea folosi asemănător linii de ieșire separate pentru a reprezenta o victorie pentru jucătorul *A*, o victorie pentru jucătorul *B* sau o remiză. Astfel, cutia ar avea șase intrări și trei ieșiri.

Folosirea a trei semnale de intrare pentru fiecare armă este un mod foarte bun de a construi funcția, dar dacă am face-o într-un calculator, probabil că am folosi o anumită codificare ce ar necesita un număr mai mic de intrări și de ieșiri. De exemplu, am putea utiliza doi biți pentru fiecare intrare și am putea folosi combinația 01 pentru a reprezenta **Foarfeca**, 10 pentru a reprezenta **Hîrtia** și 11 pentru a reprezenta **Piatra**. Am putea codifica asemănător fiecare ieșire posibilă folosind doi biți. Această codificare ar avea drept rezultat tabela mai simplă, cu trei intrări și două ieșiri, de mai jos:

	Intrări A	**Intrări B**	**Ieșiri**
	01	01	00
	01	10	10
Foarfeca = 01	01	11	01
Hîrtia = 10	10	01	01
Piatra = 11	10	10	00
Cîștigă A=10	10	11	10
Cîștigă B = 01	11	01	10
Remiză = 00	11	10	01
	11	11	00

Calculatoarele pot folosi combinații de biți pentru a reprezenta orice: numărul biților depinde de numărul mesajelor ce trebuie diferențiate. Să ne imaginăm, de exemplu, un calculator care lucrează cu literele alfabetului. Semnalele de intrare cu cinci biți pot reprezenta treizeci și două de posibilități diferite ($2^5=32$). Funcțiile din interiorul calculatorului ce lucrează cu litere folosesc cîteodată un astfel de cod, deși cel mai adesea folosesc o codificare cu șapte sau opt biți, pentru a permite reprezentarea majusculelor, a semnelor de punctuație, a cifrelor și așa mai departe. Cele mai multe calculatoare moderne folosesc reprezentarea standard a literelor alfabetului denumită ASCII (un acronim pentru American Standard Code for Information Interchange — Codul American Standard pentru Schimbul de Informații). În ASCII, șirul 1000001 reprezintă litera majusculă A, 1000010 reprezintă majuscula B, și așa mai departe. Bineînțeles, convenția este arbitrară.

Cele mai multe calculatoare au una sau mai multe convenții pentru a reprezenta numerele. Una dintre cele mai răspîndite este reprezentarea numerelor în *baza doi*, în care șirul de biți 0000000 reprezintă numărul zero, șirul 0000001 reprezintă numărul unu, șirul 0000010 reprezintă numărul doi, și așa mai departe. Descrierea calculatoarelor ca fiind „pe șaizeci și patru de biți" sau „pe treizeci și doi de biți" indică numărul de poziții binare în reprezentarea folosită de circuitele calculatorului: un calculator pe treizeci și doi de biți folosește o combinație de treizeci și doi de biți pentru a reprezenta un număr în baza doi. Sistemul de numerație cu baza doi este o convenție obișnuită, dar nimic nu impune folosirea sa. Unele calculatoare nu o folosesc deloc, iar majoritatea calculatoarelor care o folosesc repre-

zintă de asemenea numerele în alte moduri pentru scopuri diverse. De exemplu, multe calculatoare folosesc o convenție ușor diferită pentru a reprezenta numerele negative și au de asemenea o convenție numită *virgulă mobilă* pentru a reprezenta numerele care au virgulă zecimală. (Poziția virgulei „plutește" în raport cu cifrele, astfel că un număr fixat de cifre poate fi folosit pentru a reprezenta un domeniu larg de numere.) Schemele concrete de reprezentare sînt adeseori alese astfel încît să simplifice logica circuitelor ce execută operațiile aritmetice sau să facă ușoară convertirea dintr-o reprezentare în alta.

Întrucît orice funcție logică poate fi implementată ca un bloc logic boolean, este posibilă construirea unor blocuri care execută operații aritmetice, cum ar fi adunarea sau înmulțirea, folosind numere cu orice fel de reprezentare. Să ne imaginăm, de exemplu, că dorim să construim un bloc funcțional care va aduna numere pe un calculator pe opt biți. Un bloc de sumare pe opt biți trebuie să aibă șaisprezece semnale de intrare (cîte opt pentru fiecare din numerele de adunat) și opt semnale de ieșire pentru sumă. Deoarece fiecare număr este reprezentat pe opt biți, sînt două sute cincizeci și șase de combinații posibile, și fiecare poate reprezenta un număr diferit. De exemplu, am putea folosi aceste combinații pentru a reprezenta numerele între zero și două sute cincizeci și cinci, sau cele între minus o sută și plus o sută cincizeci și patru. Definirea funcției blocului ar însemna doar a scrie tabela adunării și a o converti în zerouri și unuri folosind reprezentarea aleasă. Tabelul de zerouri și unuri ar putea fi apoi convertit în blocuri **Și** și **Sau** prin metodele descrise mai sus.

Prin adăugarea a încă două intrări în bloc, am putea folosi tehnici asemănătoare pentru a construi un bloc

care nu numai că adună, dar și scade, înmulțește și împarte numerele. Cele două intrări suplimentare de control ar preciza care dintre aceste operații ar avea loc. De exemplu, pe fiecare linie a tabelei în care intrările de control ar fi 01 am preciza că ieșirea este suma numerelor de la intrare, în vreme ce pentru orice combinație în care intrările de control ar fi 10 am preciza că ieșirea este produsul, și așa mai departe. Cele mai multe calculatoare au în interior blocuri logice de acest tip, numite *unități aritmetice*.

Combinarea porților **Și** și **Sau** în conformitate cu această strategie constituie un mod de a construi orice funcție logică, dar nu este întotdeauna modul cel mai eficient. Deseori, printr-o proiectare ingenioasă, poți implementa un circuit folosind mult mai puține blocuri constructive decît ar fi necesare pentru strategia anterioară. Ar putea fi de dorit, de asemenea, folosirea altor tipuri de blocuri constructive sau proiectarea unor circuite care minimizează întîrzierea de la intrare la ieșire. Iată cîteva probleme de perspicacitate obișnuite în proiectarea circuitelor logice: cum folosești blocuri **Și** și **Invers** pentru a construi blocuri **Sau**? (Ușor.) Cum folosești mai multe blocuri **Și** și **Sau**, plus numai două blocuri **Invers**, pentru a construi funcția a trei blocuri **Invers**? (Dificil, dar posibil.) Probleme ca acestea apar pe parcursul proiectării unui calculator și sînt parte a ceea ce face procesul distractiv.

AUTOMATE CU UN NUMĂR FINIT DE STĂRI

Metodele pe care le-am prezentat pot fi folosite pentru a implementa orice funcție care rămîne constantă în timp, dar o clasă mai interesantă de funcții

sînt cele în care sînt în joc secvențe de timp. Pentru a manipula astfel de funcții, folosim un dispozitiv numit *automat cu un număr finit de stări.* Automatele cu un număr finit de stări pot fi folosite la implementarea funcțiilor ce variază în timp — funcții ce depind nu numai de intrarea actuală, ci și de istoria intrărilor precedente. De îndată ce înveți să recunoști un automat cu un număr finit de stări, le vei observa pretutindeni — în lacăte cu cifru, în pixuri, chiar și în contracte juridice. Ideea fundamentală a unui automat cu un număr finit de stări este combinarea unei tabele de căutare, construită folosind logica booleană, cu un dispozitiv de memorare. Memoria este folosită la a înmagazina un sumar al trecutului, care este *starea* unui automat cu un număr finit de stări.

Un lacăt cu cifru este un exemplu simplu de automat cu un număr finit de stări. Starea unui lacăt cu cifru este un sumar al șirului de cifre formate în lacăt. Lacătul nu memorează toate cifrele care au fost vreodată formate în el, dar memorează suficient despre cele mai recente cifre pentru a ști cînd ele formează șirul care va deschide lacătul. Un exemplu și mai simplu de automat cu un număr finit de stări este un pix escamotabil. Acest automat cu un număr finit de stări are doar două stări posibile — scos sau retras — și pixul memorează dacă butonul a fost apăsat de un număr impar sau de un număr par de ori. Toate automatele cu un număr finit de stări au o mulțime fixată de stări posibile, o mulțime de intrări admisibile care schimbă starea (apăsarea pe butonul pixului sau formarea unui număr într-un lacăt cu cifru) și o mulțime de ieșiri posibile (scoaterea sau retragerea minei pixului, deschiderea lacă-

tului). Ieșirile depind numai de stare, care în schimb depinde numai de istoria secvenței de intrări.

Un alt exemplu simplu de automat cu un număr finit de stări este un contor, cum ar fi indicatorul de la o ușă turnantă care arată numărul persoanelor ce au trecut prin ea. De fiecare dată cînd o nouă persoană trece prin ușă, starea contorului este avansată cu unu. Contorul este cu un număr *finit* de stări întrucît el poate socoti doar pînă la un anumit număr de cifre. Cînd atinge valoarea sa maximă — să spunem 999 — următoarea mișcare de avansare a indicatorului îl va readuce la zero. Kilometrajele automobilelor funcționează asemănător. O dată am condus un taxi vechi cu un kilometraj ce indica 70 000, dar nu am știut niciodată dacă taxiul parcursese șaptezeci de mii de mile, o sută șaptezeci de mii de mile sau două sute șaptezeci de mii de mile, deoarece kilometrajul avea numai o sută de mii de stări; toate aceste istorii erau echivalente în ceea ce privește kilometrajul. Tocmai de aceea matematicienii definesc deseori o stare ca „o mulțime de trecuturi echivalente".

Alte exemple familiare de automate cu un număr finit de stări includ semafoarele și panoul de comandă al unui ascensor. În aceste mașinării, șirul de stări este controlat de anumite combinații între un ceas intern și butoane de intrare, cum ar fi butonul pentru pietoni „Mergi" de la intersecții, sau butonul de apel al ascensorului, sau butoanele pentru selectarea etajului. Starea următoare a mașinii depinde nu numai de starea precedentă, ci și de semnalul sosit de la butoanele de intrare. Tranziția de la o stare la alta este determinată de o mulțime fixată de reguli, ce poate fi rezumată printr-o diagramă simplă de stare, care înfățișează tranziția între

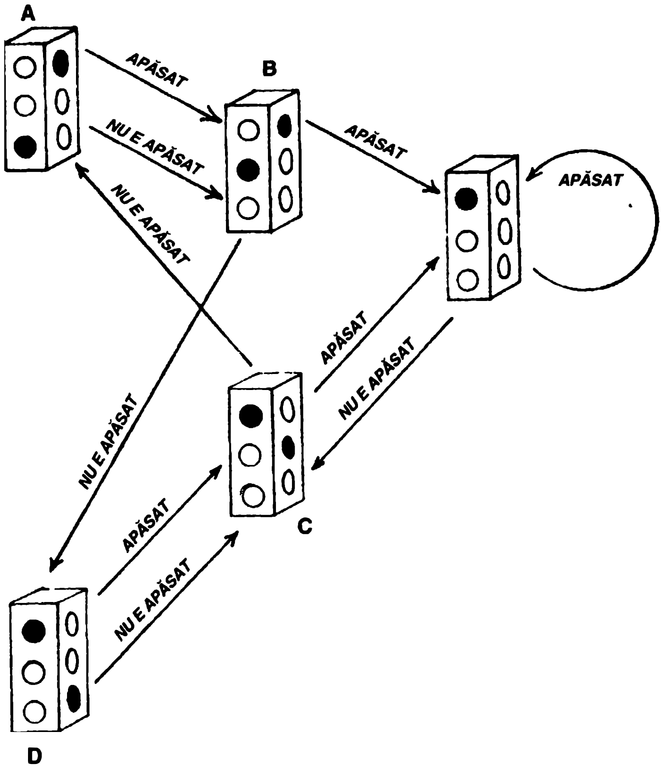


FIGURA 13 Diagrama de stare pentru un semafor

stări. Figura 13 prezintă o diagramă de stare pentru un semafor la o intersecție, care indică roșu în ambele direcții după apăsarea butonului pentru pietoni „Mergi". Fiecare imagine a semaforului reprezintă o stare și fiecare săgeată reprezintă o tranziție între stări. Tranziția depinde de faptul dacă butonul pentru pietoni a fost sau nu apăsat.

Pentru a memora stările unui automat cu un număr finit de stări avem nevoie să introducem un ultim bloc

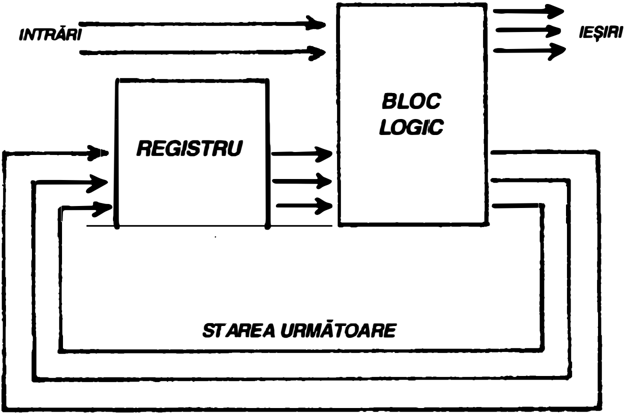


FIGURA 14 Automat cu un număr finit de stări, cu blocul logic alimentînd registrul

constructiv — un dispozitiv numit *registru* care poate fi folosit pentru a memora biți. Un registru cu *n* biți are *n* intrări și *n* ieșiri, plus o intrare temporală suplimentară care indică registrului cînd să schimbe starea. Înregistrarea unei noi informații este numită „scrierea“ stării registrului. Cînd semnalul temporal indică registrului să scrie o nouă stare, registrul își schimbă starea în conformitate cu intrările. Ieșirile registrului indică întotdeauna starea sa actuală. Registrele pot fi implementate în multe moduri, unul dintre acestea fiind folosirea unui bloc logic boolean pentru a ghida informația de stare în jurul unui cerc. Acest tip de registru este folosit deseori în calculatoarele electronice și de aceea ele pierd șirul a ceea ce fac dacă se întrerupe alimentarea cu curent.

Un automat cu un număr finit de stări constă dintr-un bloc logic boolean conectat cu un registru, așa cum este înfățișat în Figura 14. Automatul cu un număr finit

de stări avansează starea sa scriind ieșirea blocului logic boolean în registru; apoi blocul logic calculează următoarea stare pe baza intrării și a stării actuale. Starea următoare este apoi scrisă în registru în ciclul următor. Procesul se repetă la fiecare ciclu.

Funcționarea unui automat cu un număr finit de stări poate fi precizată printr-un tabel care arată, pentru fiecare stare și fiecare intrare, starea ce urmează. De exemplu, putem rezuma operarea unui dispozitiv de control al unui semafor în următorul tabel:

Intrări:			*Ieșiri:*	
Buton pt. pietoni	**Stare actuală**	**Drum principal**	**Drum secundar**	**Stare următoare**
nu e apăsat	A	roșu	verde	B
nu e apăsat	B	roșu	galben	D
nu e apăsat	C	galben	roșu	A
nu e apăsat	D	verde	roșu	C
nu e apăsat	mergi	mergi	mergi	D
apăsat	A	roșu	verde	B
apăsat	B	roșu	galben	mergi
apăsat	C	galben	roșu	mergi
apăsat	D	verde	roșu	C
apăsat	mergi	mergi	mergi	mergi

Primul pas în implementarea unui automat cu un număr finit de stări constă în generarea unui astfel de tabel. Al doilea pas este atribuirea cîte unui șir de biți diferit fiecărei stări. Cele cinci stări ale dispozitivului de control al semaforului vor necesita trei biți. (Deoarece fiecare bit dublează numărul de șiruri posibile, folosind n biți este posibil să se memoreze pînă la 2^n stări.) Înlocuind consecvent fiecare cuvînt din tabelul precedent cu un același șir binar, putem transforma tabelul într-o funcție care poate fi implementată cu logica booleană.

În sistemul semafor, un ceas controlează scrierea în registru, ceea ce face ca starea să se schimbe la intervale regulate. Un alt exemplu de automat cu un număr finit de stări care își modifică starea la intervale regulate este un ceas cu afișaj numeric. Un ceas numeric cu indicator pentru secunde poate fi în una din cele 24 × 60 × 60 = 86 400 stări ale ecranului posibil a fi afișate — cîte una pentru fiecare secundă a zilei. Mecanismul de cronometrare din interiorul ceasului îl face să își schimbe starea exact o dată la fiecare secundă. Multe alte tipuri de dispozitive numerice de calcul, inclusiv cele mai multe calculatoare de uz general, își schimbă de asemenea starea la intervale regulate, și viteza cu care avansează este numită *frecvența* mașinii. Într-un calculator, timpul nu are o curgere continuă, ci este dat de un șir fixat de tranziții între stări. Frecvența calculatorului determină viteza acestor tranziții, prin urmare corespondența între timpul fizic și cel intern. De exemplu, calculatorul portabil la care scriu această carte are frecvența de 33 MHz, ceea ce înseamnă că el își modifică starea de treizeci și trei de milioane de ori pe secundă. Calculatorul ar fi mai rapid dacă frecvența ar fi mai ridicată, dar viteza sa este limitată de timpul necesar pentru ca informația să se propage prin blocurile logice pentru a calcula următoarea stare. Pe măsura îmbunătățirii tehnologiei, logica tinde să devină mai rapidă și frecvența se mărește. În momentul în care scriu aceste cuvinte, calculatorul meu este de vîrf, dar probabil că în momentul cînd veți citi această carte calculatoarele cu frecvența de 33 MHz vor fi considerate lente. Acesta este unul din miracolele tehnologiei siliciului: pe măsură ce învățăm să producem calculatoare din ce în ce mai mici, logica devine din ce în ce mai rapidă.

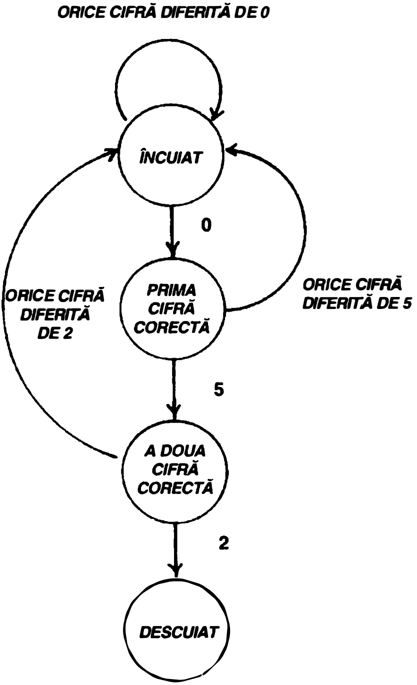


FIGURA 15 Diagrama de stare pentru un lacăt cu cifrul 0-5-2

Un motiv pentru care automatele cu un număr finit de stări sînt atît de folositoare este capacitatea lor de a recunoaște șirurile. Să considerăm un lacăt cu cifru care se deschide numai dacă i se dă secvența 0–5–2. Un astfel de lacăt, fie el mecanic sau electronic, este un automat cu un număr finit de stări cu diagrama de stare înfățișată în Figura 15.

O mașină asemănătoare poate fi construită pentru a recunoaște orice șir finit. De asemenea, automatele cu un număr finit de stări pot fi făcute să recunoască șirurile care se potrivesc unui anumit model. Figura 16 arată un automat cu un număr finit de stări care recunoaș-

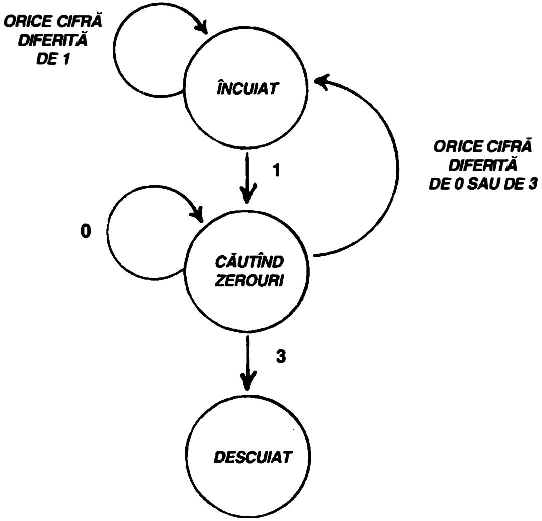


FIGURA 16 Diagrama de stare pentru a recunoaște secvențe ca 1-0-3 și 1-0-0-0-3

te orice șir care începe cu un unu urmat de oricît de multe zerouri urmate de un trei. O astfel de combinație va descuia ușa cu combinația 1-0-3, sau cu o combinație precum 1-0-0-0-3, dar *nu* cu combinația 1-0-2-3, care nu se potrivește cu modelul. Un automat cu un număr finit de stări mai complex ar putea recunoaște un model mai complicat, cum ar fi un cuvînt ortografiat incorect într-un text.

Oricît de puternice sînt, automatele cu un număr finit de stări nu sînt capabile să recunoască orice tip de model într-o secvență. De exemplu, este imposibil de construit un automat cu un număr finit de stări care va descuia un lacăt de cîte ori introduci un palindrom — un șir care este același la citirea înainte sau înapoi, precum 3-2-1-1-2-3. Aceasta deoarece palindroamele pot fi de lungime arbitrară și pentru a recu-

noaşte jumătatea a doua a unui palindrom ai nevoie să memorezi fiecare caracter din prima jumătate. Deoarece există o infinitate de posibilităţi pentru prima jumătate, ar fi nevoie de un automat cu un număr infinit de stări.

Un raţionament asemănător demonstrează imposibilitatea construirii unui automat cu un număr finit de stări care să recunoască dacă o anume propoziţie din limba engleză este corectă din punct de vedere gramatical. Să considerăm propoziţia simplă „cîinii muşcă". Înţelesul acestei propoziţii poate fi schimbat punînd altă propoziţie între substantiv şi verb; de exemplu „cîinii pe care oamenii îi supără muşcă". La rîndul său, această frază poate fi modificată punînd alte cuvinte la mijloc: „cîinii pe care oamenii cu cîini îi supără muşcă". Deşi înţelesul unor astfel de fraze poate fi exprimat mai clar şi deşi ele devin tot mai dificil de înţeles, ele sînt corecte din punct de vedere gramatical. În principiu, acest proces de a introduce propoziţii una în alta poate continua la nesfîrşit, producînd fraze absurde precum „cîinii pe care îi muşcă cîinii pe care i-au muşcat cîinii pe care îi supără cîinii muşcă". Unui automat cu un număr finit de stări îi este imposibil să recunoască dacă o astfel de frază este corectă din punct de vedere gramatical, şi din exact acelaşi motiv este dificil şi pentru o persoană: este nevoie de multă memorie pentru a urmări toţi aceşti cîini. Faptul că fiinţele umane par a avea dificultăţi cu aceleaşi feluri de fraze ca şi cele ce încurcă automatele cu un număr finit de stări a făcut ca anumite persoane să emită ipoteza că am putea avea în capul nostru ceva asemănător unui automat cu un număr finit de stări pentru înţelegerea limbajului. După cum veţi vedea în capitolul următor, sînt alte tipuri

de dispozitive de calcul ce par a se potrivi chiar mai natural cu structura recursivă a gramaticii umane.

Automatele cu un număr finit de stări mi-au fost prezentate de mentorul meu, Marvin Minsky. El mi-a prezentat următoarea problemă de perspicacitate faimoasă, numită *problema plutonului de execuție*: ești un general la comanda unui pluton de execuție de forma unui șir foarte lung de soldați. Șirul este prea lung pentru a putea striga ordinul „foc“, astfel că trebuie să dai ordin primului soldat din rînd, cerîndu-i să repete ordinul soldatului de alături, și așa mai departe. Dificultatea este că toți soldații din linie trebuie să tragă în același timp. În fundal există un răpăit constant de tobe; totuși nu poți nici măcar preciza ca toți oamenii să tragă după un anumit număr de bătăi de tobă, deoarece nu știi cîți soldați sînt aliniați. Problema este de a obține ca întregul șir să tragă simultan; o poți rezolva dînd o serie de ordine complicate care spun fiecărui soldat ce să spună vecinilor săi din fiecare parte a sa. În această problemă, soldații sînt echivalentul unui șir de automate cu un număr finit de stări, fiecare mașină modificîndu-și starea la același moment (bătaia tobei) și fiecare primind intrări de la ieșirea fiecăruia dintre vecinii săi nemijlociți. Problema este deci de a proiecta o linie de automate cu un număr finit de stări identice care vor produce ieșirea „foc“ în același moment, ca răspuns la o comandă introdusă la o extremitate a rîndului. (Automatelor de la fiecare extremitate a rîndului li se permite să fie diferite de celelalte.) Nu voi distruge enigma dîndu-i soluția, dar precizez că poate fi rezolvată folosind automate cu un număr mic de stări.

Înainte de a vă arăta cum logica booleană și automatele cu un număr finit de stări sînt combinate pentru

a produce un calculator, voi sări în această descriere de jos în sus și vă voi spune încotro ne îndreptăm. Capitolul următor începe prin a atinge unul din cele mai înalte niveluri de abstractizare în funcționarea unui calculator, care este și nivelul la care cei mai mulți programatori interacționează cu mașina.

CAPITOLUL 3

Programare

Magia exercitată de un calculator constă în capacitatea sa de a deveni aproape orice din ceea ce îți poți imagina, atîta vreme cît poți explica exact ce este acel ceva. Dificultatea constă în a explica ceea ce dorești. Cu o programare corectă, un calculator poate deveni teatru, un instrument muzical, o carte de referință, un partener la șah. Nici o altă entitate din lume, cu excepția ființei umane, nu are o capacitate de adaptare universală. În definitiv toate aceste funcții sînt implementate de blocurile logicii booleene și automatele cu un număr finit de stări descrise în capitolul precedent, dar omul care programează calculatorul rareori se gîndește la aceste elemente; în schimb programatorii lucrează cu un instrument mai convenabil numit *limbaj de programare*.

Așa cum logica booleană și automatele cu un număr finit de stări sînt blocurile constructive pentru partea „fizică“ a calculatorului (*hardware*), un limbaj de programare este o mulțime de blocuri constructive pentru partea „spirituală“ a unui calculator (*software*). Asemeni unui limbaj natural, un limbaj de programare are un vocabular și o gramatică, dar, spre deosebire de un limbaj uman, în limbajul de programare există un înțe-

les precis pentru fiecare cuvînt și fiecare propoziție. Cele mai multe limbaje de programare sînt universale, în același sens în care logica booleană este universală: ele pot fi utilizate pentru a descrie tot ceea ce poate face un calculator. Oricine a scris vreodată un program — sau a încercat să depaneze un program — știe că a-i spune unui calculator ceea ce dorești să facă nu este așa de simplu cum pare. Fiecare amănunt al operației dorite de la calculator trebuie descris precis. De exemplu, dacă ceri unui program de contabilitate să trimită clienților tăi cîte o înștiințare pentru suma pe care fiecare o datorează, atunci calculatorul va emite săptămînal o notă de plată de 0,00 $ clienților care nu datorează nimic. Dacă îi ceri calculatorului să trimită o somație clienților care nu au plătit, clienții care nu au datorii vor primi somații pînă cînd vor trimite un cec cu valoarea de 0,00 $. Programarea calculatorului înseamnă tocmai evitarea acestui gen de neînțelegeri. Arta programatorului este arta de a spune exact ceea ce vrei. În exemplul dat, înseamnă a distinge clienții care nu au trimis bani de clienții care într-adevăr *datorează* bani. Pentru a-l parafraza pe Mark Twain, diferența dintre un program corect și un program aproape corect este diferența dintre un fulger și un licurici.* Un programator iscusit este ca un poet care poate exprima acele idei pe care ceilalți le consideră inexprimabile în cuvinte. Dacă ești poet, presupui că ai în comun cu cititorul tău un bagaj de cunoștințe și o oarecare experiență. Cunoștințele și experiența pe care programatorul și calculatorul le au în comun sînt semantica limbajului de pro-

---

* Joc de cuvinte intraductibil: „the difference between lightning and a lightning bug is just a bug"; în jargonul de specialitate al programatorilor, termenul „bug" înseamnă „eroare". (*N. t.*)

gramare. Vom arăta mai tîrziu cum „știe“ calculatorul semantica limbajului de programare; în primul rînd vom discuta gramatica, vocabularul și dialectele unor astfel de limbaje.

DE VORBĂ CU CALCULATORUL

Există multe limbaje de programare diferite. Principalele motive pentru această diversitate sînt istoria, deprinderile și gustul, dar există limbaje de programare diferite și datorită faptului că ele sînt bune pentru descrierea unor tipuri diferite de lucruri. Fiecare limbaj are propria sa sintaxă. Trebuie să înveți sintaxa pentru a putea scrie într-un limbaj, dar (ca și ortografia și punctuația într-un limbaj uman) sintaxa nu este fundamentală pentru semantica sau puterea expresivă a limbajului. Ceea ce este important pentru puterea expresivă a limbajului este vocabularul — așa-numitele *primitive* ale limbajului — și modul în care primitivele pot fi combinate pentru a defini concepte noi.

Limbajele de programare descriu manipularea datelor și una din diferențele între aceste limbaje este dată de tipurile de date pe care le manipulează. Limbajele de programare din primele generații erau proiectate în primul rînd pentru a manipula numere și șiruri de caractere. Limbajele de programare ulterioare pot manipula cuvinte, imagini, sunete și chiar alte programe de calculator. Dar indiferent ce tip de date este proiectat limbajul să manipuleze, el este prevăzut cu un mod specific de a introduce datele în calculator, de a le considera separat, de a le grupa, de a le modifica, de a le compara și de a le atribui nume.

Probabil este mai ușor să ilustrăm aceste lucruri abstracte descriind un limbaj de programare concret —

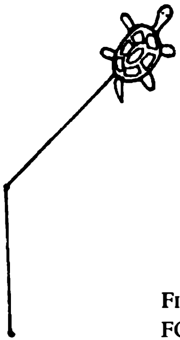


FIGURA 17
FORWARD 10, RIGHT 45, FORWARD 10

am ales Logo, care a fost proiectat de matematicianul și specialistul în educație Seymour Papert ca un limbaj de programare pentru copii. Copiii pot scrie în Logo programe care creează și manipulează imagini, cuvinte, numere și sunete. Deși limbajul este suficient de simplu pentru a fi folosit de un copil de zece ani, el conține multe din caracteristicile celor mai sofisticate limbaje de programare, inclusiv capacitatea de a scrie programe care manipulează alte programe. Logo este de asemenea un limbaj *extensibil* — adică poți folosi Logo pentru a defini cuvinte noi în Logo.

Unul din cele mai simple tipuri de programe care pot fi scrise în Logo este o procedură pentru desenarea unei imagini. Acest lucru îl faci dînd instrucțiuni unei broaște țestoase imaginare care trăiește pe ecran. Țestoasa servește drept creion, mișcîndu-se în jurul ecranului și lăsînd linii în urma sa. La pornirea calculatorului, broasca țestoasă se găsește în centrul ecranului, cu capul spre partea de sus a ecranului. Cînd copilul introduce de la tastatură comanda FORWARD 10, broasca face zece pași înainte — adică în sus — desenînd un segment lung de zece unități. Numărul 10 ce urmează comenzii FORWARD este numit *parametru*; în acest

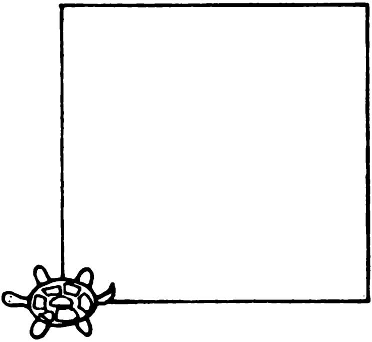


FIGURA 18 Un pătrat

caz, parametrul spune broaştei ţestoase cîţi paşi să facă. Pentru a desena o linie într-o direcţie diferită, copilul trebuie să rotească broasca ţestoasă. Comanda RIGHT 45 va orienta broasca la patruzeci şi cinci de grade (alt parametru) spre dreapta faţă de ultima sa direcţie. Următoarea comandă FORWARD, împreună cu parametrul său, va desena atunci un segment în noua direcţie.

Copilul poate folosi comenzi ca FORWARD, BACKWARD, RIGHT şi LEFT pentru a mişca broasca ţestoasă în jurul ecranului şi a desena imagini, dar acest lucru necesită apăsarea multor taste şi curînd devine plictisitor. Ceea ce face interesant limbajul este capacitatea sa de a defini cuvinte noi: de exemplu, aici vedem cum poate copilul învăţa broasca ţestoasă (programa calculatorul) pentru a desena un pătrat:

```
TO PATRAT
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
END
```

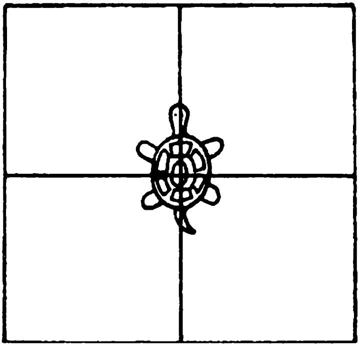


FIGURA 19 O fereastră făcută din patru pătrate

Avînd definit cuvîntul PATRAT, copilul poate desena un pătrat cu latura de zece unităţi introducînd de la tastatură o nouă comandă: PATRAT. (Nu mai e nevoie să spun că numele PATRAT este arbitrar; copilul ar fi putut la fel de bine să numească procedura CUTIE sau XYZ, iar broasca ţestoasă ar face exact acelaşi lucru. Cînd copiii descoperă acest lucru, deseori ei se amuză „păcălind" calculatorul — de exemplu, numind TRIUNGHI procedura care desenează un pătrat, şi invers.)

De îndată ce cuvîntul PATRAT a fost definit, el devine parte a vocabularului calculatorului şi poate fi folosit pentru a defini alte cuvinte. De exemplu,

```
TO FEREASTRA
PATRAT
PATRAT
PATRAT
PATRAT
END
```

Fiecare pătrat va fi desenat într-un loc diferit, deoarece procedura de desenare a pătratului lasă broasca ţestoasă rotită cu nouăzeci de grade. În termeni de specialitate, PATRAT este o *subrutină* a programului FEREASTRA, care o *apelează*. La rîndul său, subrutina PATRAT

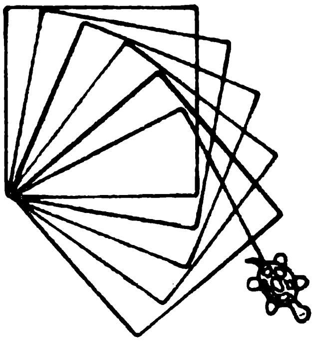


FIGURA 20 Un model, desen în curs de execuție

este definită folosind *primitivele* FORWARD și RIGHT. Cuvintele definite în Logo de utilizator pot primi de asemenea parametrii. De exemplu, un copil poate preciza diverse mărimi pentru pătrat descriind exact ce parametru determină lungimea fiecărei laturi.

```
TO PATRAT: MARIME
FORWARD: MARIME
RIGHT 90
FORWARD: MARIME
RIGHT 90
FORWARD: MARIME
RIGHT 90
FORWARD: MARIME
END
```

Cele două puncte din fața cuvîntului MARIME sînt un exemplu de sintaxă. În Logo, două puncte indică faptul că următorul cuvînt este numele unui parametru reprezentînd altceva — în acest caz, numărul ce va fi furnizat de fiecare dată cînd „apelăm" subrutina numită PATRAT. Dacă PATRAT este definită astfel, comanda PATRAT 15 va spune calculatorului să deseneze un pătrat cu latura de cincisprezece unități. Numele parametrului MARIME este din nou un nume arbitrar și are înțeles

numai în interiorul subrutinei PATRAT: dacă toate cele cinci apariții ale cuvîntului MARIME ar fi înlocuite cu X, să spunem, subrutina ar face exact același lucru.

Există alte posibilități de a scrie o subrutină pentru a desena un pătrat. De exemplu, poți da instrucțiuni broaștei țestoase să facă la stînga de patru ori sau să meargă înapoi de patru ori. Interesant este faptul că nu contează cum este definită subrutina PATRAT; ceea ce contează este ce desenează subrutina și unde lasă ea broasca țestoasă. Alte programe pot apela subrutina PATRAT indiferent de cum este ea definită și dacă este sau nu un cuvînt definit de utilizator sau unul dintre primitivele limbajului. Pentru a extinde limbajul, programatorul folosește puterea abstractizării funcționale pentru a crea noi blocuri constructive.

Unul din trucurile pe care le descoperă copiii ce folosesc Logo este acela că pot introduce un cuvînt în propria sa definiție, o practică cunoscută ca *recursivitate.* De exemplu, un copil poate genera un model circular, constînd din mai multe pătrate rotite, în felul următor:

```
TO MODEL
PATRAT
RIGHT 10
MODEL
END
```

Calculatorul execută comanda MODEL desenînd un pătrat, apoi rotind broasca țestoasă cu zece grade și desenînd alt „model" în același fel. În acest caz, definiția recursivă a „modelului" are un cusur: continuă la nesfîrșit. De fiecare dată cînd calculatorul răspunde la comanda MODEL, el desenează un pătrat și apoi el trebuie să continue cu alt model, și așa mai departe *ad infinitum* — cam ca în snoava cu un guru care afirma că

Pămîntul stă pe spatele (așa se întîmplă) unei broaște țestoase uriașe. „Și pe ce stă broasca țestoasă ?“, a întrebat un învățăcel. „Pe o altă broască țestoasă“, a răspuns guru. „Și acea broască țestoasă ?“, a întrebat învățăcelul, începînd să devină neîncrezător. „Nu are rost să întrebi“, a spus guru. „Dedesubt sînt numai broaște țestoase.“

La desenarea modelului, calculatorul parcurge același proces ca și ființele umane încercînd să își imagineze o stivă infinită de broaște țestoase uriașe, dar calculatorul nu este suficient de inteligent pentru a observa că nu ajunge nicăieri. El nu se va opri pînă nu va fi întrerupt — exemplu al unui fel obișnuit de comportare a programelor numit *buclă infinită*. Programatorii creează deseori bucle infinite din greșeală și (așa cum vom vedea) poate fi extrem de dificil de prevăzut cînd va apărea o astfel de buclă. Această buclă infinită concretă este evitată ușor scriind programul cu un parametru precizînd cîte pătrate să deseneze:

```
TO MODEL: NUMAR
PATRAT
RIGHT 10
IF: NUMAR=1 STOP ELSE MODEL: NUMĂR –1
END
```

Astfel definită, subrutina MODEL va face unul din două lucruri posibile, depinzînd de faptul dacă parametrul său este unu sau o valoare mai mare. MODEL 1 va desena numai un pătrat, dar MODEL 5, să spunem, va desena un pătrat, va roti și apoi va executa MODEL 4. MODEL 4 va desena un pătrat și apoi va executa MODEL 3, și așa mai departe, pînă la MODEL 1, care va desena un pătrat și se va opri.

Acest fel de definiție recursivă cu un parametru variabil este folositoare pentru a produce orice are o structură similară sieși. O fotografie care conține o fotografie a sa este un exemplu de structură recursivă, similară sieși; astfel de structuri sînt cunoscute în mod obișnuit ca *fractali*. În lumea reală, structuri similare lor însele nu continuă la nesfîrșit: de exemplu, fiecare ramură a unui arbore seamănă mult cu un arbore mai mic, și fiecare din aceste ramuri mai mici are ramuri care seamănă cu arbori încă și mai mici. Această recursivitate continuă pe cîteva niveluri, dar în cele din urmă ramurile sînt atît de mici încît nu mai au ramuri proprii.

Un program Logo recursiv pentru desenarea unui arbore este dat mai jos. Acesta vă poate da o idee despre cum poate un program de calculator să conțină un element de poezie — deși în acest caz tema este întru cîtva eclipsată de detaliile cerute pentru a poziționa broasca țestoasă și a o readuce la punctul de pornire. O traducere aproximativă a ceea ce spune programul ar fi: „Un arbore mare este un băț cu doi arbori mai mici în vîrf, dar un arbore mic este doar un băț." Desenul produs de programul ARBORE este arătat alături de program.

```
TO ARBORE: MARIME
FORWARD: MARIME
IF: SIZE < 1 STOP ELSE DOI-
ARBORI MARIME/2
BACK: MARIME
END
```

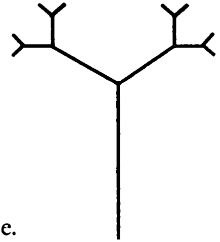


FIGURA 21 Un arbore.

```
TO: DOI-ARBORI: MARIME
LEFT 45
ARBORE: MARIME
RIGHT 90
ARBORE: MARIME
LEFT 45
END
```

Această tehnică de a defini recursiv lucruri se dovedește a fi foarte puternică. Multe tipuri de date pe care dorim să le manipulăm — în particular, înseși programele de calculator — au structuri recursive. Definițiile recursive sînt extrem de convenabile pentru a descrie operații pe date recursive. Definiția recursivă tipică are două părți — prima parte descrie ceea ce se întîmplă într-un caz particular simplu, și cea de-a doua parte descrie cum poate fi redus un caz mai complex la ceva mai simplu. În arborele recursiv, de exemplu, cazul simplu este arborele mai mic decît unu și cazul mai complicat este arborele compus dintr-un trunchi și doi arbori mici.

Un alt exemplu este definiția unui palindrom, pe care îl putem defini după cum urmează: un cuvînt este un palindrom dacă are mai puțin de două litere (cazul simplu) sau dacă prima și ultima sa literă coincid și literele dintre ele formează un palindrom (pasul recursiv). Cel mai simplu mod de a scrie un program Logo pentru recunoașterea palindroamelor ar fi să folosim acestă definiție recursivă.

Există multe alte limbaje de programare: LISP, Ada, FORTRAN, C, ALGOL și altele asemenea; cele mai multe dintre nume sînt acronime obscure (cum ar fi FORTRAN din FORmula TRANslation și LISP din LISt Processing). Deși aceste limbaje diferă de Logo prin detalii ale vocabularului și ale sintaxei, ele toate pot exprima

aceleaşi feluri de proceduri. Unele, ca FORTRAN, au capacitate limitată de a defini operaţii recursiv sau de a manipula date nenumerice. Altele, precum C sau LISP, permit programatorului să manipuleze biţii care stau la baza reprezentării datelor, ceea ce dă programatorului mai multă putere — şi mai multe prilejuri de a face greşeli. În C, de exemplu, este perfect posibil a înmulţi două caractere alfabetice; rezultatul acestei operaţii lipsite de sens va depinde de reprezentarea binară folosită de maşină. Limbaje precum LISP oferă funcţii de nivel abstract, ca şi de nivel scăzut. Aşa cum a spus cîndva un prieten al meu, informaticianul Guy Steele, „LISP este un limbaj de nivel înalt, dar în care încă poţi simţi biţii alunecîndu-ţi printre degete".

Mai recent, o nouă generaţie de limbaje a început să se dezvolte. Aceste limbaje — Small-Talk, C++, Java — sînt *orientate pe obiecte*. Ele tratează o structură de date — de exemplu, o imagine de desenat pe ecran — ca un „obiect" cu propria stare internă, cum ar fi culoarea sa ori locul unde trebuie desenat. Aceste obiecte pot primi instrucţiuni de la alte obiecte. Pentru a înţelege de ce este util acest lucru, să ne imaginăm că scrii un program pentru un joc video în care apar mingi ce ricoşează. Fiecare minge de pe ecran este definită ca un obiect diferit. Programul descrie amănunţit reguli de comportare care precizează obiectului cum să apară pe ecran, cum să se mişte, cum să sară şi cum să interacţioneze cu alte obiecte din joc. Fiecare minge va înfăţişa un comportament asemănător, dar fiecare va fi într-o stare uşor diferită, deoarece fiecare va fi în propria poziţie pe ecran şi va avea propria culoare, viteză, mărime, şi aşa mai departe.

Cel mai important avantaj al unui limbaj de programare orientat pe obiecte este acela că obiectele — de

exemplu, diverse obiecte într-un joc video — pot fi descrise amănunțit în mod independent și apoi combinate pentru a crea programe noi. Uneori scrierea unui nou program orientat pe obiecte seamănă puțin cu aruncarea unor animale într-o cușcă și urmărirea a ceea ce se întîmplă. Comportarea programului *reiese* ca un rezultat al interacțiunilor dintre obiectele programate. Din acest motiv, dar și pentru că limbajele orientate pe obiecte sînt relativ noi, ar trebui să te gîndești de două ori înainte de a scrie un program pentru un sistem în care siguranța funcționării este critică, cum ar fi un sistem ce pilotează un avion.

Învățarea unui limbaj de programare nu este nici pe departe atît de dificilă ca învățarea unui limbaj uman natural. În general, de îndată ce ai învățat două sau trei, poți deprinde altele în răstimp de cîteva ore, deoarece sintaxa este relativ simplă și vocabularul rareori depășește cîteva sute de cuvinte. Dar, ceea ce este adevărat și cu un limbaj natural, există o mare diferență între a fi capabil să înțelegi limbajul și a fi capabil să scrii bine. Fiecare limbaj de calculator are propriul Shakespeare și e o bucurie a-i citi programele. Un program de calculator bine scris are stil, finețe, chiar umor — și o claritate care rivalizează cu cele mai bune opere în proză.

FĂCÎND LEGĂTURA

Cum pot fi folosite automatele cu un număr finit de stări pentru a executa instrucțiuni scrise într-un limbaj ca Logo? Pentru a răspunde la această întrebare, revenim la un nivel mai amănunțit al discuției referitoare la logica booleană. Sînt trei pași mari în această legătură dintre automatele cu un număr finit de stări și Logo:

în primul rînd vom vedea cum o mașină cu un număr finit de stări poate fi extinsă prin adăugarea unui dispozitiv de stocare numit *memorie*, ceea ce va permite mașinii să păstreze definițiile a ceea ce i se cere să facă; *în al doilea rînd* vom vedea cum această mașină extinsă poate executa instrucțiuni scrise în *limbaj mașină*, un limbaj simplu care descrie amănunțit operațiile mașinii; și *în al treilea rînd* vom vedea cum limbajul mașină poate da instrucțiuni mașinii pentru a interpreta limbajul de programare — Logo, de exemplu. În restul capitolului este descris cu mai multe detalii cum funcționează toate acestea — cu mult mai multe detalii decît este strict necesar pentru a înțelege restul cărții. Cititorul nu trebuie să se simtă obligat să înțeleagă fiecare pas. Este important de apreciat cum se sprijină straturile de abstractizare funcțională unul de altul, așa cum este rezumat în ultimul paragraf al capitolului.

Un calculator este doar un tip special de mașină cu un număr finit de stări, conectată la o memorie. Memoria calculatorului — în fapt un aranjament de locații convenabile pentru depozitarea datelor — este constituită din *registre*, asemănătoare cu registrele care păstrează stările unui automat cu un număr finit de stări. Fiecare registru păstrează o secvență de biți numită *cuvînt*, care poate fi citită (sau scrisă) de mașina cu un număr finit de stări. Numărul de biți dintr-un cuvînt variază de la calculator la calculator, dar într-un microprocesor modern (la momentul cînd scriu această carte) este de obicei de opt, șaisprezece sau treizeci și doi. (Mărimea cuvîntului va crește probabil o dată cu îmbunătățirea tehnologiei.) O memorie tipică va avea milioane sau chiar miliarde de astfel de registre, fiecare păstrînd un singur cuvînt. La un moment dat, doar

un singur registru din memorie este accesat — adică numai datele dintr-unul din registrele memoriei vor fi citite sau scrise la fiecare ciclu al mașinii cu un număr finit de stări. Fiecare registru din memorie are o *adresă* diferită — o secvență de biți cu ajutorul cărora îl poți accesa — astfel că registrele sînt denumite *locații de memorie*. Memoria conține blocuri de logică booleană, care decodifică adresa și selectează locația pentru citire sau scriere. Dacă trebuie scrise date în această locație de memorie, aceste blocuri logice depozitează noile date în registrul adresat. Dacă registrul trebuie citit, blocurile logice dirijează datele din registrul adresat pînă la ieșirea memoriei care este conectată la intrarea mașinii cu un număr finit de stări.

Unele din cuvintele păstrate în memorie reprezintă date asupra cărora se operează, cum ar fi numere și litere. Altele reprezintă *instrucțiuni* care indică mașinii ce secvență de operații să execute. Aceste instrucțiuni sînt păstrate în limbaj mașină, care, așa cum am menționat, este mult mai simplu decît un limbaj de programare obișnuit. Limbajul mașină este interpretat direct de mașina cu un număr finit de stări. În tipul de calculator pe care îl vom descrie, fiecare instrucțiune în limbaj mașină este păstrată într-un singur cuvînt din memorie și o secvență de instrucțiuni este păstrată într-un bloc de locații de memorie numerotate consecutiv. Aceste secvențe de instrucțiuni în limbaj mașină sînt cel mai simplu fel de software dintr-un calculator.

Mașina cu un număr finit de stări execută repetat următorul șir de operațiuni: (1) *citește* o instrucțiune din memorie, (2) *execută* operațiunea specificată de acea instrucțiune și (3) *calculează* adresa instrucțiunii următoare. Secvența de stări necesară pentru a face

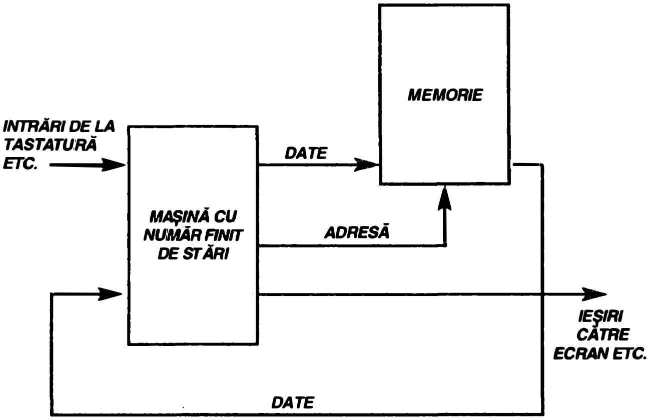


FIGURA 22 O maşină cu un număr finit de stări conectată la o memorie

acest lucru este incorporată în logica booleană a maşinii şi instrucţiunile propriu-zise sînt secvenţe specifice de biţi — secvenţe care determină maşina cu un număr finit de stări să execute diverse operaţiuni asupra datelor din memorie. De exemplu, instrucţiunea **Add** este o secvenţă unică de biţi care precizează cele două registre de memorie care trebuie adunate. După recunoaşterea acestei secvenţe, maşina cu un număr finit de stări va trece printr-un şir de stări care provoacă citirea locaţiilor de memorie ce trebuie însumate, adunarea numerelor şi scrierea sumei înapoi în memorie.

În cele mai multe calculatoare există două tipuri fundamentale de instrucţiuni: instrucţiuni de execuţie şi instrucţiuni de control. *Instrucţiunile de execuţie* mută datele în şi din memorie şi le combină pentru a executa funcţiile aritmetice şi logice. Adresele locaţiilor de memorie, sau a registrelor, sînt precizate de instrucţiunile de execuţie. În mod obişnuit, aceste instrucţi-

uni se referă direct doar la cîteva registre; referirea la alte registre se face indirect, deoarece adresele lor sînt păstrate în alte registre. De exemplu, o instrucțiune **Move** poate muta datele din registrul 1 la adresa precizată în registrul 2. Dacă registrul 2 conține o secvență de biți care specifică numărul 1 234, atunci datele vor fi mutate în registrul 1 234. Alte instrucțiuni de execuție combină datele din registrele de memorie. Există de asemenea instrucțiuni care execută funcții booleene — **Și, Sau** sau **Invers** — pe secvențele de biți din registre.

Instrucțiunile de control determină adresa următoarei instrucțiuni ce va fi încărcată; această adresă este păstrată într-un registru special numit *contor de program.* În mod normal, instrucțiunile sînt încărcate una după alta din locații de memorie consecutive, astfel încît adresa din contorul de program crește cu unu după fiecare instrucțiune succesivă. Instrucțiunile de control permit încărcarea altor numere în contorul de program, prin aceasta influențînd secvența de executat. Cea mai simplă instrucțiune de control este instrucțiunea **Jump**, care stochează o anumită adresă în contorul de program astfel încît următoarea instrucțiune va fi încărcată de la acea adresă. O variantă a instrucțiunii **Jump** este un **Jump** condiționat, care încarcă contorul de program cu o adresă diferită doar cînd a fost îndeplinită o condiție precizată — cum ar fi ca șirurile din două registre să coincidă. În cazul în care condiția nu este îndeplinită, instrucțiunea **Jump** condiționată nu va avea nici un efect și următoarea instrucțiune va fi încărcată în mod secvențial.

Dacă o aceeași secvență de instrucțiuni trebuie să fie executată în repetate rînduri, se poate folosi o instrucțiune **Jump** condiționat la sfîrșitul secvenței pentru a readuce contorul de program la începutul secvenței

de cîte ori este necesar. Această operaţiune este numită o *buclă* şi am văzut un exemplu la descrierea programării în Logo. Execuţia secvenţei va fi repetată pînă cînd nu mai este îndeplinită condiţia de care depinde saltul. Dacă o secvenţă de instrucţiuni trebuie repetată de zece ori, să spunem, unul din registrele memoriei poate fi folosit pentru a socoti numărul iteraţiilor buclei.

Calculatoarele se deosebesc după instrucţiunile pe care le recunoaşte fiecare. Proiectanţii de calculatoare consacră ani întregi pentru dezbaterea a ceea ce înseamnă o mulţime optimă de instrucţiuni. O controversă tipică se referă la meritele relative ale unui *calculator cu o mulţime redusă de instrucţiuni* (RISC), care foloseşte o mulţime de instrucţiuni simplă, minimală, şi cele ale unui *calculator cu o mulţime completă de instrucţiuni* (CISC), care foloseşte o mulţime de instrucţiuni bogată, complexă şi puternică. Această controversă are totuşi puţină importanţă pentru programator, deoarece orice mulţime de instrucţiuni rezonabilă poate simula oricare alta. Din punct de vedere istoric se pare că succesul comercial al unui tip de calculator nu are aproape nimic de-a face cu complexitatea mulţimii de instrucţiuni sau cu oricare alt detaliu intern al proiectului. De fapt, unele din calculatoarele cu cel mai mare succes — cum ar fi microprocesoarele folosite în cele mai multe calculatoare personale — sînt considerate în general de proiectanţii de calculatoare a avea mulţimi de instrucţiuni proiectate mediocru. Detaliile proiectului maşinii sînt de mică importanţă pentru utilizatorii maşinilor.

Una din cauzele pentru care complexitatea mulţimii de instrucţiuni nu contează prea mult are legătură cu subrutinele. Subrutinele permit folosirea unor secvenţe

de instrucțiuni în repetate rînduri, din multe locuri din program. De fapt, convenția de apelare a subrutinelor permite programatorului să definească instrucțiuni noi folosind secvențe de alte instrucțiuni. Programul accesează o subrutină folosind o instrucțiune **Jump** pentru a încărca în contorul de program adresa subrutinei; înainte de a face acest lucru, calculatorul salvează conținutul anterior al contorului de program într-o locație specială de memorie. La terminarea subrutinei, altă instrucțiune citește adresa de revenire și execută un salt înapoi la locația de la care a fost apelată subrutina.

Acest proces de apelare a subrutinelor poate funcționa recursiv, în sensul că secvențele de subrutine pot avea salturi la alte subrutine din interiorul lor, și așa mai departe. O subrutină se poate apela chiar pe ea însăși, într-o funcție definită recursiv. Pentru a ține evidența acestei încuibăriri de subrutine, calculatorul are nevoie de o modalitate sistematică de păstrare a adreselor de revenire, astfel că el va ști unde să revină la terminarea execuției fiecărei subrutine. El nu poate păstra pur și simplu toate adresele de revenire în aceeași locație specială de memorie, deoarece atunci cînd subrutinele sînt încuibărite trebuie păstrate mai mult de o singură adresă de revenire. În mod normal, calculatorul păstrează adresele de revenire într-un grup de locații succesive cunoscute ca *stivă*. Adresa de revenire cea mai recentă este păstrată „în vîrful stivei". Stiva de memorie este folosită întocmai ca o stivă de farfurii: obiectele sînt întotdeauna depuse sau luate din vîrf — un sistem de depozitare „ultimul sosit, primul servit" care funcționează perfect pentru păstrarea adreselor de revenire pentru subrutine încuibărite, întrucît o subrutină nu este niciodată terminată înainte ca toate subrutinele încuibărite în ea să fie terminate.

Unele subrutine sînt atît de folositoare încît sînt încărcate permanent în calculator. Această mulțime de subrutine este numită *sistem de operare*. Exemple de subrutine folositoare din sistemul de operare sînt cele de scriere sau citire a caracterelor introduse de la tastatură, sau cele care desenează linii pe ecran, sau cele ce interacționează în alt mod cu utilizatorul. Sistemul de operare al calculatorului este determinant pentru cea mai mare parte a aspectului interfeței cu utilizatorul. De asemenea, el determină interfața dintre calculator și fiecare program ce este rulat, întrucît subrutinele din sistemul de operare înzestrează programul cu un set de operații mai bogate și mai complexe decît instrucțiunile limbajului mașină.

De fapt, atîta timp cît aceeași secvență de biți produce același efect, programatorului nu îi pasă dacă o funcție este implementată de hardware-ul calculatorului sau de software-ul sistemului de operare. Același program, rulînd pe două tipuri diferite de calculatore, ar putea să execute o operație aritmetică în hardware într-un caz, iar în celălalt caz să o execute prin intermediul unei subrutine din sistemul de operare. În mod asemănător, sistemul de operare al unui tip de calculator îi poate permite să execute în același fel, dar poate nu la aceeași viteză, întregul set de instrucțiuni al altui tip de calculator. Producătorii de calculatoare folosesc uneori o astfel de execuție pentru a face ca modelul mai nou al calculatoarelor lor să acționeze ca și modelele anterioare, astfel încît programele mai vechi să poată rula fără modificări.

În mod normal, sistemul de operare conține toate subrutinele care realizează intrarea-ieșirea — adică operații permițînd unui program să interacționeze cu lumea exterioară. Această interacțiune este efectuată prin conectarea anumitor locații din memoria calculatorului

cu dispozitive de intrare, cum ar fi o tastatură sau un șoricel (*mouse*), și cu dispozitive de ieșire, cum ar fi un terminal cu ecran video. De exemplu, bara de spațiu de la tastatură ar putea fi conectată cu registrul 23 din memorie, astfel încît data citită de la adresa 23 este 1 dacă bara de spațiu este apăsată și 0 în caz contrar. Un alt registru din memorie ar putea controla culoarea afișată într-un anumit punct de pe ecran. Dacă fiecare punct de pe ecran afișează data păstrată într-o locație diferită de memorie, calculatorul poate desena pe ecran orice model, pur și simplu scriind în memorie modelul potrivit.

Cu excepția mecanismelor sale de intrare-ieșire, calculatorul pe care tocmai l-am descris este pur și simplu o mașină cu un număr finit de stări legată la o memorie. Amîndouă aceste elemente pot fi construite în întregime din registre și blocuri logice booleene, folosind tehnicile descrise în capitolele unu și doi. Mașina cu un număr finit de stări care controlează calculatorul este complicată, dar în principiu nu este diferită de mașina cu un număr finit de stări care controlează un semafor. Proiectarea mașinii înseamnă pur și simplu a trece prin detaliile datelor din memorie adresa și secvența de stări pentru fiecare instrucțiune de executat și apoi a converti această tabelă de stare în logică booleană. Să ne amintim că, întrucît și mașina cu un număr finit de stări și memoria sînt constituite din registre și blocuri logice, ambele pot fi realizate prin mai multe tehnologii: electronică, hidraulică, tije culisante.

TRADUCEREA LIMBAJULUI

Am stabilit, așadar, un lanț de legături între tehnologie și instrucțiuni. Dar cum execută instrucțiunile un

program scris într-un limbaj — Logo, de exemplu —, cînd limbajul este scris în cuvinte iar instrucțiunile sînt șiruri de biți ? Răspunsul este: traducerea necesară este executată de calculatorul însuși.

Procesul de traducere realizat de calculator este asemănător procesului prin care un traducător uman răbdător și meticulos ar traduce un document scris într-o limbă cu care nu e familiarizat, fiind dat un dicționar explicativ scris în însăși acea limbă. Traducătorul poate căuta în dicționar înțelesul oricărui cuvînt necunoscut, și dacă termenii din definiția dată în dicționar îi sînt de asemenea necunoscuți, poate căuta și aceste cuvinte. Acest proces continuă pînă cînd traducătorul ajunge la o definiție formulată în cuvinte al căror înțeles îl cunoaște. În această analogie, dicționarul traducătorului (adică al calculatorului) este programul, iar cuvintele cunoscute calculatorului sînt primitivele, menționate mai înainte, din limbajul de programare. Aceste primitive sînt definite direct, ca secvențe simple de instrucțiuni mașină. De exemplu, cînd calculatorul caută definiția primitivei FORWARD din limbajul Logo, el găsește secvența de instrucțiuni mașină care va desena linia potrivită pe ecran.

Pentru a înțelege cum traduce un calculator primitivele din Logo în limbaj mașină, este util să înțelegem convențiile pe care le folosește calculatorul pentru a reprezenta în memoria sa programele Logo. O posibilitate de a stoca un program Logo în memoria unui calculator constă în a folosi o secvență de caractere în locații de memorie alăturate, cu fiecare caracter păstrat într-o singură locație. Calculatorul păstrează în memoria sa un catalog cu adresele șirurilor de instrucțiuni corespunzînd fiecărui nume de comandă. Acest catalog este păstrat în memorie ca o listă de numere

alături de adresele lor. Calculatorul este în stare să găsească locația unui obiect cu un nume dat căutînd în catalog numele și găsind adresa corespunzătoare. Cînd i se cere unui calculator să execute o anume comandă, el caută în catalog numele pentru a afla unde este păstrată definiția sa.

O parte din acest proces de căutare și de găsire a șirurilor corespunzătoare în limbaj mașină poate fi făcută înainte ca programul să fie executat. Astfel se economisește timp, deoarece, dacă programul urmează să fie executat de mai multe ori, nu are sens căutarea aceluiași lucru de mai multe ori. Cînd cea mai mare parte a acțiunii de conversie este făcută dinainte, procesul de traducere se numește *compilare*, iar programul care realizează compilarea se numește *compilator*. Dacă cea mai mare parte a acțiunii este făcută în timpul execuției programului, procesul este numit interpretare, iar programul este numit *interpretor*. Nu există o linie de demarcație riguroasă între cele două.

BUN VENIT LA IERARHIE

Acum sîntem în măsură să rezumăm funcționarea unui calculator, de sus și pînă jos. Cei mai mulți cititori vor fi pierdut șirul amănuntelor, dar *să ne amintim că nu este important să ne amintim cum funcționează fiecare pas*! Ceea ce este important de ținut minte este ierarhia de abstractizări funcționale.

Activitatea executată de un calculator este precizată de un *program*, care este scris într-un *limbaj de programare*. Acest limbaj este convertit în șiruri de instrucțiuni în *limbaj mașină* de *interpretoare* și *compilatoare*, printr-un set predefinit de subrutine numit *sistem de*

operare. Instrucțiunile, care sînt păstrate în *memoria* calculatorului, definesc operațiunile de executat asupra datelor, care sînt de asemenea păstrate în memoria calculatorului. Un *automat cu un număr finit de stări* încarcă și execută aceste instrucțiuni. Instrucțiunile, ca și datele, sînt reprezentate prin secvențe de *biți*. Atît automatul cu un număr finit de stări, cît și memoria, sînt constituite din *registre* și *blocuri logice booleene*, iar ultimele se bazează pe *funcții logice* simple, cum ar fi **Și, Sau** sau **Invers**. Aceste funcții logice sînt implementate în *circuite cu comutatoare*, care sînt fie *în serie*, fie *în paralel*, și aceste comutatoare controlează o substanță fizică, cum ar fi apa sau electricitatea, care este folosită pentru a trimite de la un comutator la altul unul din două semnale posibile: 1 sau 0. Aceasta este ierarhia de abstractizări care face calculatoarele să funcționeze.

CAPITOLUL 4

Cît de universale sînt mașinile Turing?

Care sînt limitele a ceea ce poate face un calculator? Toate calculatoarele trebuie să fie constituite din logică booleană și registre, sau ar putea fi de alte feluri, poate chiar mai puternice? Aceste întebări ne conduc la subiectele cele mai interesante din punct de vedere filozofic ale acestei cărți: mașinile Turing, calculabilitatea, sistemele haotice, teorema lui Gödel de incompletitudine și calculul cuantic — subiecte aflate în centrul celor mai multe discuții referitoare la ceea ce pot face și nu pot face calculatoarele.

Deoarece calculatoarele pot face unele lucruri care par foarte asemănătoare gîndirii umane, oamenii se tem deseori că ele amenință poziția noastră unică de ființe raționale și sînt unii care caută semne de încurajare în demonstrații matematice ale limitelor calculatoarelor. În istoria umană au existat controverse asemănătoare. Cîndva s-a considerat important ca Pămîntul să fie în centrul Universului și poziția noastră imaginată în centru era considerată emblematică pentru valoarea noastră. Descoperirea faptului că noi nu ocupam o poziție centrală — că planeta noastră era doar una dintre cele cîteva planete învîrtindu-se în jurul Soarelui — a fost profund îngrijorătoare pentru mulți oameni ai acelui

timp, iar implicațiile filozofice ale astronomiei au devenit subiect de dezbateri înfierbîntate. O controversă asemănătoare s-a născut cu privire la teoria evoluționistă, care a apărut de asemenea ca o amenințare la unicitatea omenirii. La rădăcina acestor crize filozofice din trecut a fost o judecată deplasată asupra sursei valorii umane. Sînt convins că majoritatea discuțiilor filozofice contemporane despre limitele calculatoarelor sînt bazate pe o judecată greșită asemănătoare.

MAȘINI TURING

Ideea centrală în teoria calculului este aceea a unui *calculator universal* — adică un calculator suficient de puternic pentru a simula orice alt dispozitiv de calcul. Calculatorul de uz general descris în capitolele precedente este un exemplu de calculator universal; de fapt, cele mai multe calculatoare pe care le întîlnim în viața de fiecare zi sînt calculatoare universale. Cu un program potrivit, suficient timp și destulă memorie, orice calculator universal poate simula orice alt tip de calculator, sau (după cîte știm) orice alt dispozitiv pentru prelucrarea informației.

O consecință a acestui principiu al universalității este aceea că singura diferență importantă între puterea a două calculatoare este dată de viteza lor și de mărimea memoriilor lor. Calculatoarele pot diferi prin feluritele dispozitive de intrare-ieșire conectate la ele, dar aceste așa-numite periferice nu sînt caracteristici esențiale ale unui calculator, nu mai mult decît mărimea, costul sau culoarea carcasei lor. În privința a ceea ce pot face, toate calculatoarele (și toate celelalte tipuri de dispozitive universale de calcul) sînt fundamental identice.

Ideea unui calculator universal a fost recunoscută și descrisă în 1937 de matematicianul britanic Alan Turing. Turing, ca și mulți alți pionieri ai calculului, era interesat în problema construirii unei mașini care ar putea gîndi și a inventat o schemă pentru o mașină de calcul de uz general. Turing se referea la construcția sa imaginară ca la o „mașină universală", deoarece în acea vreme cuvîntul „calculator" încă însemna „persoană care efectuează calcule".

Pentru a descrie o mașină Turing, să ne imaginăm un matematician efectuînd calcule pe un sul de hîrtie. Să ne imaginăm mai departe că sulul are o lungime infinită, așa încît nu trebuie să ne îngrijoreze faptul că nu am avea loc pentru a scrie. Matematicianul va fi capabil să rezolve orice problemă solubilă prin calcul, indiferent de cît de multe operații ar fi necesare, deși ar putea să-i ia un interval de timp excesiv de lung. Turing a arătat că orice calcul care poate fi executat de un matematician inteligent poate fi de asemenea executat de un funcționar nătîng, dar meticulos, care urmează o mulțime de reguli simple pentru a citi și scrie informația de pe sul. De fapt, a arătat că funcționarul uman poate fi înlocuit de un automat cu un număr finit de stări. Automatul cu un număr finit de stări examinează la un moment dat un singur simbol de pe sul, așa încît cel mai bine este să considerăm sulul ca o bandă îngustă de hîrtie, cu un singur simbol pe fiecare linie.

Combinația unui automat cu un număr finit de stări cu o bandă de lungime infinită o numim astăzi *mașină Turing*. Banda unei mașini Turing este asemănătoare și îndeplinește aproape aceleași funcții ca memoria unui calculator modern. Tot ceea ce face un automat cu un număr finit de stări este să citească și să scrie un simbol pe bandă și să o deplaseze înainte sau înapoi în con-

formitate cu o mulţime fixată şi simplă de reguli. Turing a arătat că orice problemă de calcul ar putea fi rezolvată scriind simboluri pe banda unei maşini Turing — simboluri ce ar preciza nu numai problema, ci şi metoda de rezolvare a ei. Maşina Turing calculează răspunsul mişcîndu-se înainte şi înapoi pe bandă, citind şi scriind simboluri, pînă ce soluţia este scrisă pe bandă.

Mi se pare dificil să ne gîndim la construcţia concretă a lui Turing. Pentru mine, calculatorul obişnuit, care are o memorie în loc de o bandă, este un exemplu de maşină universală mai uşor de înţeles. Îmi este mai uşor să înţeleg, de exemplu, cum un calculator obişnuit poate fi programat pentru a simula o maşină Turing, decît invers. Ceea ce mă uimeşte nu este atît construcţia imaginară a lui Turing, cît presupunerea sa că există doar un tip de maşină de calcul universală. După cîte ştim, nici un dispozitiv construit în universul fizic nu poate avea mai multă putere de calcul decît o maşină Turing. Într-o exprimare mai precisă, orice calcul care poate fi executat de orice dispozitiv fizic de calcul poate fi executat de orice calculator universal, atîta vreme cît acesta din urmă are suficient timp şi destulă memorie. Această formulare este remarcabilă, sugerînd că un calculator universal cu programare potrivită ar putea fi capabil să simuleze funcţionarea creierului uman.

NIVELURI DE PUTERE

Cum poate fi adevărată ipoteza lui Turing ? Desigur, s-ar putea ca alt fel de calculator să fie mai puternic decît acelea pe care le-am descris. Pe de o parte, calculatoarele pe care le-am discutat pînă acum au fost binare, adică

ele reprezintă totul numai cu zerouri și unuri. Nu ar fi mai puternic un calculator dacă ar putea reprezenta lucrurile în termenii unei logici cu trei stări, precum **Da, Nu** și **Poate**? Nu, nu ar fi. Știm că un calculator cu trei stări nu ar fi capabil să facă mai mult decît un calculator cu două stări, deoarece poți simula pe fiecare folosindu-l pe celălalt. Cu un calculator cu două stări poți duplica orice operație care poate fi executată de un calculator cu trei stări codificînd fiecare din cele trei stări ca o pereche de biți — 00 pentru **Da**, să spunem, 11 pentru **Nu** și 01 pentru **Poate.** Pentru orice funcție posibilă în logica trivalentă există o funcție corespunzătoare în logica bivalentă care operează pe această reprezentare. Ceea ce nu înseamnă că aceste calculatoare cu trei stări nu ar putea avea unele avantaje practice asupra calculatoarelor cu două stări: de exemplu, ar putea folosi mai puține circuite și de aceea ar putea fi mai mici sau mai ieftin de produs. Dar putem afirma cu certitudine că ele nu ar fi capabile să facă ceva nou. Ele ar fi doar încă o versiune a unei mașini universale.

Un raționament asemănător este valabil pentru calculatoarele cu patru stări, sau calculatoarele cu cinci stări, sau calculatoarele cu orice număr finit de stări. Dar ce este cu calculatoarele care calculează cu semnale analogice — adică semnale cu un număr infinit de valori posibile? Să ne imaginăm, de exemplu, un calculator care folosește un domeniu continuu de tensiuni pentru a indica numerele. În loc de numai două sau trei sau cinci mesaje posibile, fiecare semnal ar putea transporta un număr infinit de mesaje posibile, corespunzînd domeniului continuu de tensiuni. De exemplu, un calculator analogic ar putea reprezenta un număr între zero și unu printr-o tensiune electrică între zero volți și un volt. Fracția ar putea fi reprezentată cu orice

nivel de precizie, indiferent de numărul zecimalelor, folosind exact tensiunea corespunzătoare.

Există calculatoare care reprezintă cantități prin astfel de semnale analogice și de fapt primele calculatoare au funcționat în acest fel. Ele sînt numite *calculatoare analogice* pentru a le deosebi de calculatoarele numerice despre care am discutat, care au un număr finit de mesaje posibile în fiecare semnal. S-ar putea presupune că aceste calculatoare analogice ar fi mai puternice, deoarece ele pot reprezenta un continuum de valori, în vreme ce calculatoarele numerice pot reprezenta datele doar ca numere discrete. Totuși, acest avantaj aparent dispare la o examinare mai atentă. Un continuum veritabil este irealizabil în lumea fizică.

Problema cu calculatoarele analogice este că semnalele lor pot atinge doar un grad limitat de precizie. Orice tip de semnal analogic — electric, mecanic, chimic — va conține o anumită cantitate de zgomot; adică la un anumit nivel al rezoluției, semnalul va fi esențialmente aleator. Un semnal analogic nu poate să nu fie afectat de numeroase surse de zgomot necunoscute și irelevante: de exemplu, un semnal electric poate fi afectat de mișcarea întîmplătoare a moleculelor în interiorul unui fir, sau de un cîmp magnetic creat cînd se aprinde lumina în camera alăturată. Într-un circuit electric foarte bun, acest zgomot poate fi făcut foarte mic — să spunem a milioana parte din mărimea semnalului însuși — dar el va exista totdeauna. În timp ce există o infinitate de niveluri posibile de semnal, doar un număr finit de niveluri reprezintă distincții semnificative — adică reprezintă informații. Dacă o parte dintr-un milion dintr-un semnal este zgomot, atunci există cam numai un milion de distincții semnificative în semnal; prin urmare, informația din semnal poate fi reprezen-

tată de un semnal numeric care folosește douăzeci de biți (2^{20}=1 048 578). Dublarea numărului de distincții semnificative într-un calculator analogic ar necesita ca totul să fie de două ori mai precis, în timp ce într-un calculator numeric ai putea dubla numărul distincțiilor semnificative prin adăugarea unui singur bit. Cele mai bune calculatoare analogice au precizia de mai puțin de treizeci de biți. Deoarece calculatoarele numerice reprezintă deseori numerele folosind treizeci și doi sau șaizeci și patru de biți, ele pot genera în practică un număr mult mai mare de distincții semnificative decît pot genera calculatoarele analogice.

Unii ar putea obiecta că, deși zgomotul unui calculator analogic ar putea să nu fie semnificativ, nu este neapărat nefolositor. Cu siguranță se pot imagina calcule care sînt ajutate de prezența zgomotului. De exemplu, mai tîrziu vom descrie calcule care necesită numere aleatoare. Dar și un calculator numeric poate genera zgomot aleator dacă acest caracter aleatoriu este necesar într-un calcul.

NUMERE ALEATOARE

Cum poate genera un calculator numeric ceva cu caracter aleatoriu? Poate un sistem determinist, precum un calculator, produce un șir de numere cu adevărat aleatoare? Într-un sens strict, răspunsul este NU, deoarece tot ceea ce face un calculator numeric este determinat de proiectul său și de intrările sale. Dar același lucru ar putea fi spus și despre o roată de ruletă — la urma urmelor, locul în care va ateriza bila în final este determinat de proprietățile fizice ale bilei (masa, viteza sa) și ale roții ce se învîrte. Dacă am cunoaște proiec-

tul exact al aparatului şi „intrările“ exacte ce determină învîrtirea roţii şi lansarea bilei, am putea prezice numărul pe care ar ateriza bila. Rezultatul pare aleatoriu, deoarece el nu prezintă nici un model evident şi în practică este dificil a-l prezice.

Asemeni roţii ruletei, un calculator poate produce un şir de numere care este aleator în acelaşi sens. De fapt, folosind un model matematic, calculatorul ar putea simula proprietăţile fizice ale roţii ruletei şi ar putea simula aruncarea bilei sub un unghi uşor diferit de fiecare dată pentru a produce fiecare număr din şir. Chiar dacă unghiurile sub care calculatorul lansează bila simulată urmează consecvent un model, dinamica simulată a roţii ar transforma aceste diferenţe minuscule în echivalentul unui şir imprevizibil de numere. Un astfel de şir de numere este numit şir *pseudoaleator*, deoarece el pare aleator numai unui observator care nu cunoaşte cum a fost calculat. Şirul produs de un generator de numere pseudoaleatoare poate trece toate testele statistice normale de stabilire a caracterului aleatoriu.

O roată de ruletă este un exemplu de ceea ce fizicienii numesc un *sistem haotic* — un sistem în care o schimbare mică în condiţiile iniţiale (lansarea bilei, masa bilei, diametrul roţii, şi aşa mai departe) poate produce o schimbare majoră în starea în care ajunge sistemul (numărul rezultat). Această noţiune de sistem haotic ajută la explicarea modului în care o mulţime deterministă de interacţiuni poate produce rezultate imprevizibile. Într-un calculator există moduri mai simple de a produce un şir pseudoaleatoriu decît simularea unei roţi de ruletă, dar toate acestea sînt asemănătoare din punct de vedere conceptual cu acest model.

Calculatoarele numerice sînt previzibile şi imprevizibile în exact acelaşi sens ca şi restul lumii fizice.

Ele urmează legi deterministe, dar aceste legi au consecințe complicate, care sînt extrem de dificil de prevăzut. De multe ori este nepractic să ghicim ceea ce urmează a face calculatoarele înainte de a o face. După cum e adevărat în cazul sistemelor fizice, nu trebuie mult pentru a complica un calcul. În calculatoare, sistemele haotice — sisteme ale căror ieșiri depind cu o mare sensibilitate de condițiile inițiale — sînt standardul.

CALCULABILITATE

Cu toate că un calculator universal poate calcula tot ceea ce poate fi calculat de orice alt dispozitiv de calcul, există unele lucruri care pur și simplu sînt imposibil de calculat. Desigur, nu este posibil a calcula răspunsuri pentru întrebări vagi, precum „Care este semnificația vieții ?“, sau întrebări pentru care nu avem date, precum „Care este numărul cîștigător la loteria de mîine ?“. Dar există și probleme de calcul bine definite care sînt imposibil de rezolvat. Astfel de probleme sînt numite *necalculabile*.

Ar trebui să vă previn că problemele necalculabile nu prea apar în practică. De fapt, este dificil de găsit exemple de probleme necalculabile, bine definite și pe care fiecare dorește să le rezolve. Un exemplu rar de problemă bine definită, folositoare, dar necalculabilă, este problema *opririi*. Să ne imaginăm că vreau să scriu un program pentru calculator care va examina un alt program de calculator și va determina dacă acel program se va opri sau nu în cele din urmă. Dacă programul examinat nu are bucle sau apeluri recursive la subrutine, el trebuie neapărat să se termine în cele din urmă, dar dacă programul are astfel de construcții, el

poate rula la nesfîrșit. Se dovedește că nu există algoritm pentru a examina un program și a determina dacă el este sau nu infectat fatal cu o buclă fără sfîrșit. Mai mult, nu-i vorba că nimeni nu a descoperit încă un astfel de algoritm, ci un astfel de algoritm nu este posibil. Problema opririi este necalculabilă.

Pentru a înțelege de ce, să ne imaginăm pentru un moment că există un astfel de program, numit Test-de-Oprire, care primește ca intrare programul de testat. (Tratarea unui program ca date poate părea ciudată, dar este perfect posibilă deoarece un program, ca orice altceva, poate fi reprezentat prin biți.) Aș putea insera programul Test-de-Oprire ca o subrutină în alt program, numit Paradox, care va executa Test-de-Oprire asupra lui Paradox însuși. Să ne imaginăm că am scris programul Paradox astfel încît indiferent de ceea ce stabilește Test-de-Oprire, Paradox va face opusul. Dacă Test-de-Oprire stabilește că Paradox se va opri în cele din urmă, atunci Paradox este programat să intre într-un ciclu infinit. Dacă Test-de-Oprire stabilește că Paradox va continua la infinit, atunci Paradox este programat să se oprească. Deoarece Paradox contrazice Test-de-Oprire, Test-de-Oprire nu funcționează cu Paradox; de aceea, el nu funcționează pe orice program. Și deci nu poate exista un program care să calculeze funcția de oprire.

Problema opririi, care a fost imaginată de Alan Turing, este importantă în special ca un exemplu de problemă necalculabilă, și cele mai multe probleme necalculabile ce apar în practică sînt asemănătoare sau echivalente cu ea. Dar incapacitatea unui calculator de a rezolva problema opririi nu este o slăbiciune a calculatorului, întrucît problema opririi este intrinsec insolubilă. Nu se poate construi o mașină care să rezolve proble-

ma opririi și, din cîte știm, nu există nimic din ceea ce poate executa orice alt calculator și care să nu poată fi executat de o mașină universală. Clasa problemelor care sînt calculabile de un calculator numeric include în mod evident orice problemă care este calculabilă de orice fel de dispozitiv. (Această ultimă afirmație este uneori numită teza lui Church, după unul din contemporanii lui Turing, Alonzo Church. Matematicienii s-au gîndit la calcul și logică de secole, dar — în unul din cele mai uimitoare exemple de sincronism în știință — Turing, Church și alt matematician britanic, pe nume Emil Post, au inventat cu toții în mod independent ideea calculului universal, aproximativ în același timp. Ei aveau moduri foarte diferite de a-l descrie, dar toți și-au publicat rezultatele în 1937, pregătind scena pentru revoluția calculatoarelor ce va urma curînd.)

Altă funcție necalculabilă, înrudită îndeaproape cu problema opririi, este problema de a decide dacă orice aserțiune matematică dată este adevărată sau falsă. Nici pentru această problemă nu există algoritm care să o poată rezolva — o consecință a teoremei lui Gödel de incompletitudine, care a fost demonstrată de Kurt Gödel în 1931, cu puțin înainte ca Turing să fi descris problema opririi. Teorema lui Gödel a constituit un șoc pentru mulți matematicieni, care pînă atunci au presupus în general că *orice* enunț matematic poate fi dovedit adevărat sau fals. Teorema lui Gödel afirmă că în orice sistem matematic necontradictoriu, suficient de puternic pentru a exprima aritmetica, există enunțuri despre care nu se poate demonstra nici că sînt adevărate, nici că sînt false. Matematicienii considerau că treaba lor era de a demonstra validitatea sau falsitatea unor enunțuri, și teorema lui Gödel a demonstrat că „misiunea“ lor era imposibilă în anumite cazuri.

Unii matematicieni şi filozofi au atribuit proprietăţi aproape mistice teoremei de incompletitudine a lui Gödel. Cîţiva cred că teorema dovedeşte că intuiţia umană depăşeşte cumva puterea unui calculator — acele fiinţe umane putînd fi capabile să „intuiască“ adevăruri a căror validitate sau falsitate nu poate fi demonstrată de maşini. Acest argument este atrăgător din punct de vedere emoţional şi a fost folosit de filozofii cărora nu le place să fie comparaţi cu calculatoarele. Dar raţionamentul este greşit. Indiferent dacă oamenii fac sau nu cu succes salturi intuitive ce nu pot fi făcute de calculatoare, teorema lui Gödel de incompletitudine nu furnizează nici un motiv pentru a crede că există enunţuri matematice care pot fi demonstrate de un matematician, dar nu pot fi demonstrate de un calculator. Din cîte ştim, orice teoremă ce poate fi demonstrată de o fiinţă umană poate fi demonstrată şi de un calculator. În faţa problemelor necalculabile pe care calculatoarele nu le pot rezolva, oamenii sînt la fel de neputincioşi.

Deşi se insistă mult pentru a veni cu exemple concrete de probleme necalculabile, se poate demonstra uşor că majoritatea funcţiilor matematice posibile sînt necalculabile. Aceasta deoarece orice program poate fi formulat într-un număr finit de biţi, în vreme ce formularea unei funcţii necesită de obicei o infinitate de biţi, astfel încît există cu mult mai multe funcţii decît programe. Să considerăm un tip de funcţie matematică, cea care transformă un număr în altul — să spunem cosinus sau logaritm. Matematicienii pot defini tot felul de funcţii bizare de acest tip: de exemplu funcţia care transformă orice număr scris în baza zece în suma cifrelor sale. Din cîte ştiu eu, această funcţie este nefolositoare, dar un matematician ar considera-o ca o funcţie

legitimă pur şi simplu deoarece ea transformă orice număr în exact un alt număr. Se poate demonstra matematic că există infinit mai multe funcţii decît programe. Prin urmare, pentru cele mai multe funcţii nu există un program corespunzător care să le poată calcula. Calculul efectiv implică dificultăţi de tot felul (inclusiv calculul cu entităţi infinite şi distingerea între grade diferite de infinitudine!), dar concluzia este corectă: din punct de vedere statistic, cele mai multe dintre funcţiile matematice sînt necalculabile. Din fericire, aproape toate aceste funcţii necalculabile sînt inutile şi realmente toate funcţiile pe care am dori să le calculăm sînt calculabile.

CALCUL CUANTIC

Aşa cum am notat mai devreme, şirurile de numere pseudoaleatoare produse de calculatoare par a fi aleatoare, dar există un algoritm pe care se bazează şi care le generează. Dacă ştii cum este generat un şir, în mod necesar el este previzibil şi nu aleatoriu. Dacă am fi avut vreodată nevoie de un şir de numere intrinsec imprevizibile, ar fi trebuit să extindem maşina noastră universală cu un dispozitiv nedeterminist pentru generarea fenomenelor cu caracter aleatoriu.

Se poate imagina un astfel de dispozitiv pentru producerea fenomenelor cu caracter aleatoriu ca fiind un fel de ruletă electronică, dar, aşa cum am văzut, un astfel de dispozitiv nu este cu adevărat aleator, datorită legilor fizice. Singurul mod pe care îl cunoaştem pentru a realiza efecte cu adevărat imprevizibile este de a ne baza pe mecanica cuantică. Spre deosebire de fizica clasică a roţii de ruletă, în care efectele sînt deter-

minate de cauze, mecanica cuantică produce efecte care sînt pur probabiliste. Nu există nici un mod de a prezice, de exemplu, cînd un atom dat de uraniu se va dezintegra devenind atom de plumb. Prin urmare, se poate folosi un contor Geiger pentru a genera şiruri de date pur aleatoare — ceva în principiu imposibil de făcut cu un calculator universal.

Legile mecanicii cuantice ridică un număr de întrebări despre calculatoarele universale, întrebări la care nimeni nu a răspuns încă. La prima vedere ar părea că mecanica cuantică se potriveşte bine cu calculul numeric, deoarece cuvîntul „cuantic" transmite esenţialmente aceeaşi noţiune ca şi cuvîntul „numeric". Asemeni fenomenelor numerice, fenomenele cuantice există numai în stări discrete. Din punct de vedere cuantic, natura analogică şi (aparent) continuă a lumii fizice — fluxul electricităţii, de exemplu — este o iluzie provocată de faptul că noi vedem lucrurile la o scară mare în loc să le privim la scara atomică. Aspectul pozitiv al mecanicii cuantice este că la scară atomică totul este discret, totul este numeric. O sarcină electrică va conţine un anumit număr de electroni şi nu există ceva precum o jumătate de electron. Aspectul negativ este că regulile ce determină modul în care obiectele interacţionează la această scară contrazic intuiţia.

De exemplu, noţiunile noastre de bun-simţ ne spun că un lucru nu poate fi în două locuri în acelaşi timp. În lumea mecanicii cuantice aceasta nu este întru totul adevărat, deoarece în mecanica cuantică nimic nu se poate afla exact într-un anume loc. O singură particulă subatomică există pretutindeni în acelaşi timp, şi noi doar avem şansa de a observa o astfel de particulă mai degrabă într-un loc decît într-un altul. Pentru cele mai multe scopuri este potrivit să ne închipuim o parti-

culă aflîndu-se acolo unde o observăm, dar pentru a explica toate efectele observate trebuie să recunoaștem că particula este în mai mult decît un singur loc. Pentru aproape oricine, inclusiv pentru mulți fizicieni, acest concept este dificil de înțeles.

Am putea trage foloase din efectele cuantice pentru a construi un tip mai puternic de calculator? Pînă acum, această întrebare rămîne fără răspuns, dar sînt sugestii că așa ceva este posibil. Atomii par a fi capabili de a soluționa ușor anumite probleme, cum ar fi felul în care se fixează reciproc — probleme foarte dificil de rezolvat cu un calculator obișnuit. De exemplu, cînd doi atomi de hidrogen se leagă de un atom de oxigen pentru a forma o moleculă de apă, acești atomi „calculează“ cumva că unghiul dintre cele două legături trebuie să fie de o sută șapte grade. E posibil să calculăm aproximativ acest unghi pornind de la principiile mecanicii cuantice și folosind un calculator numeric, dar durează mult, cu atît mai mult cu cît calculul este mai precis. Și totuși, fiecare moleculă dintr-un pahar cu apă este capabilă să execute acest calcul aproape instantaneu. Cum poate fi o singură moleculă incomparabil mai rapidă decît un calculator numeric?

Motivul pentru care unui calculator îi ia atît de mult timp să soluționeze această problemă de mecanică cuantică este acela că, pentru a da un răspuns exact, calculatorul ar trebui să considere o infinitate de configurații posibile pentru molecula de apă. Calculul trebuie să admită faptul că atomii ce compun molecula pot fi în toate configurațiile deodată. De aceea calculatorul nu poate decît aproxima răspunsul într-un interval finit de timp. Un mod de a explica cum poate face molecula de apă același calcul este să ne-o imaginăm încercînd toate configurațiile posibile simultan — cu alte

cuvinte, folosind prelucrarea paralelă. Am putea utiliza această capacitate de calcul simultan a obiectelor mecanicii cuantice pentru a produce un calculator mai puternic? Nimeni nu știe cu siguranță.

De curînd au apărut cîteva indicații tulburătoare că am putea fi capabili să construim un calculator cuantic care profită de un fenomen numit *suprapunere coerentă*. Într-un sistem cuantic, atunci cînd două particule interacționează, destinele lor pot deveni legate într-un mod cu totul diferit de tot ceea ce vedem în lumea fizică clasică: cînd măsurăm o caracteristică a uneia dintre ele, este influențat ceea ce măsurăm la cealaltă, chiar dacă particulele sînt separate din punct de vedere fizic. Einstein a numit acest efect, care nu implică o întîrziere în timp, „acțiune fantomatică la distanță“ și e de notorietate faptul că nu se împăca cu posibilitatea ca lumea să funcționeze în acest mod.

Un calculator cuantic ar profita de suprapunerea coerentă: un registru cuantic de memorie de un bit ar memora nu doar un zero sau un unu; el ar înmagazina o suprapunere de multe zerouri și unuri. Această situație este analogul unui atom aflat în multe locuri deodată: un bit care este în mai multe stări (1 sau 0) deodată. Aceasta diferă de o stare intermediară între zero și unu, deoarece fiecare dintre 1 și 0 suprapuse pot fi în stare de suprapunere coerentă cu alți biți din interiorul calculatorului cuantic. Cînd doi astfel de biți cuantici sînt combinați într-un bloc logic cuantic, fiecare din stările lor suprapuse poate interacționa în moduri diferite, producînd o mulțime chiar și mai bogată de suprapuneri coerente. Cantitatea de calcule ce pot fi realizate de un singur bloc logic cuantic este foarte mare, poate chiar infinită.

Teoria pe care se bazează calculul cuantic este bine stabilită, dar încă sînt probleme cu folosirea ei. Pe de o parte, cum putem folosi tot acest calcul pentru a calcula ceva util ? Fizicianul Peter Shor a descoperit de curînd un mod de a folosi aceste efecte cuantice — în principiu, cel puțin — pentru a efectua anumite calcule dificile și importante, precum factorizarea numerelor mari, iar lucrările sale au reînnoit interesul pentru calculatoarele cuantice. Dar există încă multe dificultăți. O problemă este aceea că biții unui calculator cuantic trebuie să rămînă în starea de suprapunere coerentă astfel încît calculul să se poată desfășura, dar cea mai mică tulburare — o rază cosmică întîmplătoare, să spunem, sau poate chiar și zgomotul intrinsec al vidului însuși — poate distruge suprapunerea cuantică. (Da, în mecanica cuantică chiar și vidul face lucruri stranii.) Această pierdere a suprapunerii coerente, numită *decoerență*, s-ar putea dovedi a fi un călcîi al lui Ahile al calculatoarelor cuantice. Mai mult, metoda lui Shor pare a funcționa numai pentru o clasă anume de calcule care pot profita de o operație rapidă numită transformare Fourier generalizată. Problemele care se încadrează în această categorie s-ar putea dovedi ușor de calculat pe o mașină Turing clasică; dacă este așa, ideile cuantice ale lui Shor ar fi echivalente cu un program pe un calculator obișnuit.

Dacă va deveni posibil pentru calculatoarele cuantice să caute o infinitate de posibilități deodată, atunci ele ar fi fundamental mai puternice, în mod calitativ, decît mașinile de calcul obișnuite. Cei mai mulți oameni de știință ar fi surprinși dacă mecanica cuantică ar reuși să furnizeze un fel de calculator mai puternic decît o mașină Turing, dar știința progresează printr-o serie de surprize. Dacă sperați să fiți surprinși de un tip nou

de calculator, mecanica cuantică este un domeniu care merită atenția.

Aceasta ne conduce înapoi la chestiunile filozofice atinse la începutul capitolului — anume la relația dintre calculator și creierul uman. Desigur că se poate concepe, după cum cel puțin un bine cunoscut fizician a emis ipoteza (în huiduielile majorității colegilor săi) că creierul uman folosește avantajele oferite de efectele mecanicii cuantice. Însă nu există nici o dovadă că lucrurile stau astfel. Desigur, fizica unui neuron depinde de mecanica cuantică la fel ca fizica unui tranzistor, dar nu există nici o dovadă că prelucrarea neurală are loc la nivelul mecanicii cuantice, opus nivelului clasic; adică nu există nici o dovadă că mecanica cuantică este necesară pentru explicarea gîndirii umane. Din cîte știm, toate proprietățile calculatorii relevante ale unui neuron pot fi simulate pe un calculator obișnuit. Și dacă într-adevăr este așa, atunci este de asemenea posibilă simularea unei rețele de zeci de miliarde de astfel de neuroni, ceea ce înseamnă, în schimb, că se poate simula creierul pe o mașină universală. Chiar dacă se dovedește că avantajele calculului cuantic sînt folosite de creier, vom învăța probabil cum să construim dispozitive care folosesc avantajele acelorași efecte — caz în care tot va fi posibilă simularea creierului uman cu o mașină.

Limitările teoretice ale calculatoarelor nu furnizează nici o linie utilă de demarcație între ființele umane și calculatoare. Din cîte știm, creierul este un fel de calculator, iar gîndirea este doar un calcul complex. Poate că această concluzie vă sună neplăcut, dar în concepția mea ea nu diminuează cu nimic miracolul sau valoarea gîndirii umane. Afirmația că gîndirea este un calcul complex este ca și afirmația, făcută cîndva de un biolog,

că viața este o reacție chimică complexă: ambele afirmații sînt adevărate, și totuși ele încă pot fi considerate incomplete. Ele identifică în mod corect componentele, dar ignoră misterul. Pentru mine, viața și gîndirea devin cu atît mai minunate prin însăși înțelegerea faptului că ambele rezultă din părți simple, ușor de înțeles. Nu mă simt degradat de înrudirea mea cu mașina lui Turing.

CAPITOLUL 5

Algoritmi și euristică

În primii mei ani de studenție la Institutul de Tehnologie din Massachusetts, unul din colegii de cameră avea cîteva zeci de perechi de șosete, fiecare pereche cu un model sau o culoare ușor diferite. El amîna de multe ori spălatul pînă rămînea complet fără șosete curate, astfel că de cîte ori le spăla avea sarcina deloc simplă de a le împerechea. Iată cum proceda: întîi scotea la întîmplare o șosetă din maldărul de lenjerie curată, apoi scotea altă șosetă la întîmplare și o compara cu prima să vadă dacă se potrivesc. Dacă nu, arunca înapoi a doua șosetă și scotea alta. Continua să facă asta pînă găsea perechea, apoi urma aceeași succesiune cu o nouă șosetă. Cum trebuia să caute prin multă lenjerie, procesul era foarte lent — îndeosebi la început, cînd erau mult mai multe șosete de examinat înainte de a apărea o potrivire.

El studia matematica și se pare că urma un curs de informatică. Într-o zi, cînd a adus coșul cu rufe în camera noastră, a anunțat: „Am hotărît să folosesc un algoritm mai bun pentru a-mi împerechea șosetele.". Voia să spună că era pe cale de a folosi un procedeu fundamental diferit. A scos prima șosetă și a pus-o pe masă, apoi a scos următoarea șosetă și a comparat-o cu prima;

deoarece nu se potriveau, a așezat-o lîngă prima. Acum de fiecare dată cînd scotea o șosetă o compara cu cele aflate pe masă, într-un rînd tot mai lung. Cînd găsea o pereche, le strîngea laolaltă și le arunca în sertarul cu ciorapi. Dacă nu, adăuga șoseta desperecheată la rînd. Folosind această metodă, el a fost în stare să împerecheze șosetele într-un timp mult mai scurt decît cel necesar anterior. Părinții săi, care plăteau o sumă mare de bani pentru studiile sale universitare, ar fi fost mîndri să știe că el folosea în mod practic recentele sale cunoștințe.

GARANȚIA ALGORITMICĂ

Cu sau fără șosete, un *algoritm* este o procedură sigură, garantată pentru a atinge un scop precizat. Numele „algoritm" vine de la numele matematicianului arab al-Khwarizmi, care a notat o mulțime de algoritmi în secolul al nouălea. Cuvîntul „algebră" provine de fapt de la *al jabr* („transpunerea"), termen aflat în titlul uneia din cărțile sale. Mulți dintre algoritmii lui al-Khwarizmi sînt folosiți și astăzi. El i-a descris, desigur, în limba arabă, ceea ce ar explica, poate, de ce această limbă a cîștigat reputația de limbă a descîntecelor magice. (Chiar s-a sugerat că formula magică „abracadabra" este o denaturare pentru numele său complet, Abu Abdullah abu Jafar Muhammad ibn Musa al-Khwarizmi.)

Algoritmii pentru calculatoare sînt de obicei formulați ca programe. Deoarece termenul desemnează succesiunea de operații și nu modul concret în care ele sînt descrise, un același algoritm poate fi formulat în limbaje formale diferite sau poate fi implementat în

hardware prin conectarea registrelor și porților logice adecvate.

De obicei, mai mulți algoritmi pot calcula același rezultat. Ca în exemplul cu șosetele, algoritmi diferiți au nevoie de intervale de timp diferite pentru a duce la bun sfîrșit aceeași sarcină. Anumiți algoritmi pot de asemenea să ofere alt fel de avantaje: ei pot folosi doar o cantitate mică din memoria calculatorului sau pot necesita un model deosebit de simplu de comunicare, ușor de implementat în hardware. Un algoritm bun și unul prost diferă ca viteză și necesar de memorie printr-un factor de ordinul miilor sau chiar al milioanelor. Uneori descoperirea unui nou algoritm permite soluționarea unor probleme anterior de nerezolvat.

Deoarece un algoritm poate fi implementat în multe moduri diferite și poate fi aplicat la probleme de dimensiuni diferite, nu poți aprecia cît de rapid este un algoritm prin măsurarea timpului scurs pînă se obține soluția la problema ta concretă. Timpul va varia în funcție de metoda de implementare și de dimensiunea problemei. De obicei descriem viteza unui algoritm după cum crește, în funcție de dimensiunea problemei, timpul necesar pentru a rezolva sarcina. În exemplul împerecherii șosetelor, cea mai mare parte a timpului era consumată cu scoaterea șosetelor din coș, deci putem compara cei doi algoritmi de împerechere raportînd numărul șosetelor extrase de fiecare dată la numărul total de ciorapi. Să presupunem că sînt *n* șosete în coș. În primul algoritm, pentru găsirea unei perechi este necesară scoaterea și punerea înapoi în medie a jumătate din numărul șosetelor rămase, astfel că numărul extragerilor este proporțional cu pătratul numărului șosetelor. În analiza algoritmilor, de obicei nu ne interesează valoarea exactă a constantei de proporționalitate;

mai degrabă spunem că algoritmul e de *ordin* n^2, ceea ce înseamnă că pentru probleme mari timpul necesar pentru rezolvarea completă crește ca pătratul dimensiunii problemei. Aceasta înseamnă că dacă sînt de zece ori mai multe șosete, primul algoritm va dura de o sută de ori mai mult, încît nu e un algoritm foarte bun pentru a împerechea un mare număr de ciorapi. În al doilea algoritm, în schimb, fiecare din cele n șosete este scoasă numai o singură dată, deci algoritmul este de ordin n. Cînd se folosește al doilea algoritm pentru a sorta de zece ori mai multe șosete, treaba va dura doar de zece ori mai mult.

Una din cele mai mari bucurii pe care le oferă programarea calculatoarelor este descoperirea unui algoritm nou, mai rapid, mai eficient pentru a realiza ceva — îndeosebi dacă multe persoane respectate au dat soluții mai puțin bune. Informaticienii pot cîștiga faimă și admirație — cel puțin printre informaticieni — descoperind un algoritm mai rapid pentru o problemă comună. Deoarece unui algoritm prost îi pot fi necesare săptămîni întregi pentru a rezolva o problemă pe care un algoritm bun o soluționează în cîteva minute, forma clasică pentru a fi cu un pas înaintea competitorilor constă în a scrie un program nou și a obține răspunsul corect în vreme ce programul inferior al colegului este încă în curs de execuție.

Adeseori, cel mai bun algoritm nu este evident. Să considerăm problema ordonării crescătoare a unui pachet de cărți numerotate consecutiv. Un mod de a proceda este să începem să căutăm în tot pachetul cartea cu cel mai mic număr. Scoatem această carte, care devine prima din pachetul sortat. Apoi căutăm cartea cu cea mai mică valoare dintre cele rămase. La rîndul său, aceasta este eliminată și depusă deasupra primei cărți.

Procesul este repetat pînă se epuizează cărțile nesortate și pachetul este aranjat în ordine crescătoare. Acest procedeu necesită căutare prin întregul pachet de fiecare dată cînd o carte este extrasă. Fiind *n* cărți, fiecare necesitînd *n* comparații, timpul de execuție al algoritmului este de ordin n^2.

Dacă știm că numerele cărților sînt de la unu la *n*, le putem sorta printr-o metodă diferită — una care folosește o definiție recursivă, ca în procedura Logo pentru desenarea unui arbore, descrisă în capitolul trei. Pentru a sorta recursiv un pachet de cărți, parcurgem o dată pachetul, punînd cărțile cu valoarea sub medie în jumătatea inferioară a pachetului și lăsînd în jumătatea superioară cărțile cu valoarea mai mare decît media. Apoi sortăm cele două jumătăți ale pachetului folosind același algoritm. Aplicarea recursivă a aceluiași algoritm fiecărei jumătăți de pachet implică aplicarea sa succesivă fiecărei jumătăți a jumătăților de pachet și așa mai departe. La fiecare pas recursiv, numărul cărților de sortat se înjumătățește; aplicarea recursivă ia sfîrșit cînd există o singură carte — atunci ea este deja sortată. Deoarece acest algoritm implică înjumătățirea repetată a numărului cărților pînă se examinează una singură, timpul său de execuție este proporțional cu numărul care reprezintă de cîte ori această înjumătățire poate avea loc — adică logaritm în baza doi din numărul cărților. Așadar, ordinul acestui algoritm este *n* log *n*. (Dacă nu vă amintiți ce sînt logaritmii, nu are nici o importanță. Cu toții sînt numere mici, deci pot fi ignorate fără grijă.)

Există un algoritm recursiv chiar mai elegant, care nu folosește numerotarea consecutivă a cărților și care ar fi util de exemplu la ordonarea alfabetică a unui mare număr de cărți de vizită. Acest algoritm, numit *sortarea*

prin interclasare, este mai dificil de înțeles, dar este atît de frumos încît nu rezist tentației de a-l descrie. Algoritmul de sortare prin interclasare depinde de faptul că este ușor să îmbini într-o singură stivă ordonată două stive deja ordonate trăgînd succesiv cărțile de deasupra uneia sau alteia din stive; această procedură de interclasare este o subrutină a algoritmului, iar algoritmul funcționează astfel: dacă pachetul are o singură carte, aceasta este deja sortată. Altminteri înjumătățesc pachetul și aplic recursiv algoritmul de sortare prin interclasare; după ce sortez fiecare jumătate, le combin folosind procedura descrisă mai sus. Și cu asta am sfîrșit. (Dacă vi se pare prea simplu, încercați cu cîteva cărți. Începeți cu opt.) Algoritmul de sortare prin interclasare este un bun exemplu pentru eleganța și puterea aproape misterioase ale recursivității.

Un algoritm de sortare în numai $n \log n$ pași, așa cum este sortarea prin interclasare, este destul de rapid. De fapt, este aproape cel mai rapid algoritm posibil. Demonstrarea acestei afirmații depășește scopul acestei cărți, dar raționamentul pe care se bazează demonstrația este interesant. Poate fi demonstrată calculînd numărul ordonărilor posibile a n cărți. Din acest calcul rezultă că trebuie cunoscuți $n \log n$ biți de informație pentru a pune cărțile în ordinea corectă. Deoarece fiecare comparație a două cărți produce doar un bit de informație (prima carte fie este mai mare decît a doua, fie nu este), sortarea a n numere necesită cel puțin $n \log n$ comparații, deci în acest caz algoritmul de sortare prin interclasare este aproape la fel de bun ca oricare altul. Au fost scrise cărți întregi avînd ca subiect alegerea algoritmului optim de sortare. În multe situații, cînd se impun anumite condiții sortării, sau cînd sînt disponibile cunoștințe suplimentare despre

obiectele de sortat, cel mai rapid algoritm de sortare posibil rămîne necunoscut. Totuși, comparativ cu alte probleme pentru care am dori să proiectăm algoritmi, sortarea este considerată o sarcină relativ ușoară.

Un exemplu de sarcină dificilă este *problema comis-voiajorului*. Să ne imaginăm un comis-voiajor ce trebuie să viziteze *n* orașe. Fiind cunoscută distanța dintre oricare două orașe, în ce ordine ar trebui să le viziteze comis-voiajorul pentru a minimiza distanța totală parcursă ? Nimeni nu cunoaște un algoritm de ordin n^2 sau de ordin n^3 sau de ordin *n* la orice putere și care să rezolve sarcina. Cel mai bun algoritm cunoscut este de ordin 2^n, ceea ce înseamnă că timpul necesar crește exponențial cu dimensiunea problemei. Dacă adăugăm încă zece orașe la itinerariul comis-voiajorului, problema devine de o mie de ori mai dificilă (2^{10}=1 024). Dacă adăugăm încă treizeci de orașe, problema devine cam de un miliard de ori mai dificilă (2^{30} este aproximativ 10^9). Algoritmii exponențiali nu sînt de mult folos cînd problemele devin mari, dar ei sînt cei mai buni algoritmi cunoscuți pentru problema comis-voiajorului. Cel mai rapid calculator din lume, lucrînd miliarde de ani, nu ar avea timp suficient pentru a găsi cel mai bun traseu pentru doar cîteva mii de orașe.

Problema comis-voiajorului poate părea lipsită de importanță, dar se dovedește că ea este echivalentă cu o sumedenie de alte probleme — așa-numitele *probleme NP-complete* (NP înseamnă „polinomial nedeterministic") — a căror rezolvare ar fi foarte utilă. O soluție rapidă la problema comis-voiajorului ar conduce imediat la o soluție pentru aceste probleme suplimentare: de exemplu, anumite coduri folosite pentru protejarea informațiilor secrete ar deveni foarte ușor de

spart. Fiecare dintre utilizatorii acestor coduri pariază că niciodată nu va fi găsit un algoritm rapid pentru soluționarea problemei comis-voiajorului. Probabil este un pariu cîștigător.

Nici un progres tehnic previzibil în construirea calculatoarelor nu va ajuta la rezolvarea problemei comis-voiajorului, întrucît chiar și un calculator de un miliard de ori mai rapid va fi pus în încurcătură prin adăugarea cîtorva orașe. Algoritmii exponențiali sînt pur și simplu prea lenți pentru a fi folosiți pentru probleme de dimensiuni mari. Inventarea unui nou algoritm ar fi ceva cu totul diferit: nimeni nu a demonstrat vreodată că un algoritm rapid pentru problema comis-voiajorului *nu poate* exista. Studiul algoritmilor a cunoscut progrese semnificative în doar cîteva decenii, și găsirea unui algoritm rapid pentru problema comis-voiajorului — sau demonstrarea faptului că unul rapid nu există — rămîne una din pietrele filozofale ale calculului.

REZOLVÎND APROAPE ÎNTOTDEAUNA

Oricît de dificilă ar fi problema comis-voiajorului, ea nu este una dintre cele mai dificile probleme de rezolvat pe un calculator. Sînt cunoscute cîteva probleme care necesită timp chiar mai mult decît exponențial pentru a fi rezolvate. Așa cum s-a discutat în capitolul precedent, există probleme necalculabile despre care știm că nici un algoritm nu le poate rezolva. Chiar și atunci cînd există algoritmi pentru anumite probleme, ei nu constituie neapărat cea mai bună abordare. Un algoritm, prin definiție, este garantat pentru a rezolva sarcina, dar adesea această garanție a succesului vine

la un preţ prea mare. În multe cazuri este mai practic să folosim o procedură care produce răspunsul corect numai *aproape întotdeauna*. Deseori, „aproape întotdeauna" este suficient de bine. O regulă care tinde să dea răspunsul corect, dar fără nici o garanţie pentru aceasta, este numită *euristică*. Adeseori este mai practic să folosim o euristică decît un algoritm: de exemplu, există multe euristici eficiente pentru problema comis-voiajorului — proceduri ce vor produce foarte rapid un traseu aproape optim. Aceste euristici găsesc de obicei cel mai bun traseu, deşi nu există garanţia absolută că aşa ceva se întîmplă. Un comis-voiajor real va fi probabil mai mulţumit cu o euristică bună, rapidă, decît cu un algoritm încet.

Un exemplu simplu de folosire a euristicilor este jocul de şah. Un programator talentat care este un şahist de valoare medie poate scrie un program de jucat şah care să învingă sistematic programatorul. Un astfel de program nu este un algoritm, pentru că nu există garanţia că va cîştiga. Euristicile fac alegeri bazate pe cunoştinţe; euristicile bune de obicei fac alegerile corecte. Unele dintre cele mai impresionante performanţe ale calculatoarelor sînt mai degrabă rezultatul unor euristici decît ale unor algoritmi. (Filozofii au scris foarte multe prostii despre „limitările calculatoarelor", cînd de fapt se referă la limitările algoritmilor.)

Un program bun de jucat şah poate fi scris pe baza următoarelor euristici:

1. estimează tăria relativă a poziţiei fiecărui jucător prin numărarea pieselor de fiecare fel rămase pe tablă,
2. mută astfel încît după cîteva mutări viitoare să te afli în cea mai puternică poziţie posibilă,
3. aşteaptă-te ca adversarul să adopte o strategie asemănătoare cu a ta.

Fiecare dintre aceste reguli este doar o aproximare pentru strategia ideală şi este posibil de imaginat situaţii în care fiecare este de fapt greşită. Tăria relativă a poziţiei unui jucător, de exemplu, depinde nu numai de numărul pieselor, dar şi de poziţia lor. O poziţie bună poate fi deseori mai avantajoasă decît o piesă în plus. Chiar dacă aşa stau lucrurile, prima euristică este în general corectă; în majoritatea cazurilor, e mai bine să ai mai multe piese. Încă înainte de apariţia calculatoarelor, jucătorii de şah au dezvoltat o metodă simplă de a cuantifica numeric tăria relativă a poziţiilor celor doi jucători, atribuind un punct pentru un pion, trei puncte pentru un nebun, cinci pentru o tură şi aşa mai departe, şi folosind scorul total al pieselor rămase fiecărui jucător ca o măsură a tăriei.

Pe baza acestor reguli euristice poţi scrie un program pentru jocul de şah care va urmări toate direcţiile plauzibile de joc pentru următoarele cîteva mutări. Desigur, ar fi preferabil dacă programul ar considera *toate* direcţiile de joc, plauzibile sau nu, *toate* complet pînă la sfîrşitul partidei. Acest lucru a fost uşor pentru jocul „trei-în-linie“, dar în ceea ce priveşte şahul se dovedeşte a nu fi practic, chiar şi pentru cele mai rapide calculatoare. Într-o poziţie tipică din partea de mijloc a unei partide de şah, un jucător are aproximativ treizeci şi şase de mutări regulamentare posibile, fiecare dintre acestea conducînd la treizeci şi şase de răspunsuri posibile ale adversarului. Întrucît o partidă de şah durează în medie mai mult de optzeci de mutări, calculatorul ar avea de căutat ceva de ordinul a 36^{80} posibilităţi, sau aproximativ 10^{124} posibilităţi. O astfel de căutare nu ar putea fi realizată de cele mai rapide calculatoare moderne în sute de ani. Problema este că direcţiile posibile de joc cresc exponenţial cu numărul de mutări;

astfel, este nepractic a anticipa mai mult de cinci pînă la zece mutări — și tocmai de aceea calculatoarele folosesc regulile euristice pe care tocmai le-am enunțat pentru a evalua mutările lor.

Să acceptăm pentru moment că a doua regulă euristică este corectă și să cădem de acord că cea mai bună linie de joc este aceea care optimizează poziția unui jucător după cîteva mutări viitoare. Să mai precizăm că un program pentru jocul de șah va anticipa șase mutări. În conformitate cu prima regulă euristică, programul va evalua tăria pozițiilor celor doi jucători după a șasea mutare, socotind numărul pieselor de fiecare culoare rămase pe tablă și notîndu-le cu ajutorul sistemului de punctaj pe care l-am descris. Tăriile relative ale celor doi jucători în fiecare poziție considerată vor fi judecate după diferența dintre aceste note.

Fiind date aceste ipoteze, care este cel mai bun mod în care calculatorul își alege următoarea mutare ? Nu este suficient pentru calculator să aleagă o mutare conducînd la cea mai favorabilă secvență de șase mutări ulterioare, deoarece fiecare mutare în această secvență va fi determinată de adversar. În loc de așa ceva, trebuie să presupunem că adversarul va încerca întotdeauna să aleagă o linie de joc care va favoriza poziția relativă a adversarului; această ipoteză este conținută în a treia regulă euristică. Pentru a anticipa linia de joc a adversarului, calculatorul trebuie să se pună el însuși în locul adversarului. Calculatorul alege o mișcare după evaluarea fiecărei mutări regulamentare ce poate fi făcută de partea sa — albă, să spunem. Procedura pentru evaluarea unei mutări posibile pentru alb depinde de invocarea unei proceduri pentru evaluarea unei mutări posibile pentru negru, și viceversa. De fapt, cal-

culatorul urmărește fiecare linie posibilă de joc în șase mutări, situîndu-se alternativ în locul negrului și albului. Programul încearcă mutările pe o tablă de joc imaginară în interiorul memoriei calculatorului, într-un mod foarte asemănător celui în care un maestru de șah își imaginează liniile de joc „în cap“. Programele pentru evaluarea pozițiilor albului și negrului se apelează recursiv unul pe celălalt ca subrutine. Recursivitatea ia sfîrșit după șase niveluri, atunci cînd calculatorul evaluează nota după numărarea pieselor.

Cele mai multe programe de jucat șah încorporează reguli euristice suplimentare pentru a abandona liniile de joc neplauzibile și pentru a căuta mai adînc în ramurile ce implică schimburi de piese. Există de asemenea sisteme mai sofisticate pentru evaluarea pozițiilor fără căutare — de exemplu, sisteme care acordă puncte pentru păstrarea controlului asupra centrului tablei de șah sau pentru protejarea regelui. Fiecare din aceste reguli euristice este doar o presupunere suplimentară și fiecare poate îmbunătăți căutarea în anumite situații, cu prețul unor eventuale greșeli potențiale în alte situații. Cu diverse rafinări, această procedură fundamentală de căutare stă la baza aproape a fiecărui program de jucat șah. Ea este eficientă deoarece profită de viteza cu care calculatorul examinează multe milioane de linii alternative de joc. Între aceste multe milioane de posibilități există deseori o variabilitate care va surprinde programatorul sau chiar un șahist uman experimentat. Această capacitate de a surprinde este ceea ce permite mașinii să joace o partidă mai bine decît programatorul.

Mașinile ce joacă șah au un rol îndelungat și uneori neonorabil în istoria calculului. Inventatorul maghiar

din secolul al optsprezecelea Wolfgang von Kempelen a captat imaginaţia lumii cu un automat pentru jocul de şah sub forma unui turc mecanic cu turban. După cum s-a dovedit, maşina funcţiona numai pentru că un jucător de şah pitic era ascuns în interiorul ei. În 1914, inginerul spaniol Luis Torres y Quevedo a construit un dispozitiv mecanic care juca o partidă simplificată de şah fără ajutorul unei fiinţe umane ascunse, iar la sfîrşitul anilor 1940 Claude Shannon a descris modul în care poate fi programat un calculator cu un set de reguli euristice asemănătoare celor enunţate aici. Totuşi, aceasta era cu mulţi ani înainte ca ordinatoarele să fi fost suficient de rapide pentru a juca o partidă de şah la nivel decent, ceea ce a consolat nu puţini filozofi care susţineau că şahul este un exemplu pentru puterile unice ale minţii umane. Calculatoarele moderne, folosind aceeaşi euristică, pot învinge acum cei mai buni jucători de şah din lume (mărturie stă victoria din 1997 a calculatorului Deep Blue, produs de IBM, asupra lui Garry Kasparov), aşa încît filozofii şi-au mutat raţionamentul în alte domenii.

Căutarea euristică simplă funcţionează pentru că există un număr relativ mic de răspunsuri ce trebuie luate în consideraţie pentru fiecare mutare. La table, unde există chiar mai puţine răspunsuri posibile la fiecare mutare, maşinile bazate pe euristică au început să învingă cei mai buni jucători umani încă din anii 1960. Pe de altă parte, oamenii încă domină în jocul chinezesc de go, deoarece tabla mai mare (19 × 19) permite mult mai multe mutări posibile. (Prefer să joc go decît şah tocmai pentru că o căutare este mai puţin folositoare; ceea ce face ca nerăbdarea mea să fie un dezavantaj mai mic.)

PEISAJE DE ADECVARE

Folosirea euristicii pentru a căuta într-o mulţime de posibilităţi este răspîndită în programarea calculatoarelor şi are aplicaţii mult mai importante decît jocurile. Acesta este adeseori modul în care calculatoarele găsesc soluţii „creatoare" la probleme — de obicei probleme despre ale căror soluţii se ştie că sînt printre elementele unei mulţimi finite, dar mari, de posibilităţi, mulţime numită *spaţiul* căutării. În şah, spaţiul căutării este mulţimea tuturor liniilor posibile de joc; în problema comis-voiajorului, acest spaţiu al căutării constă din toate rutele posibile ce unesc oraşele de pe lista vînzătorului. Întrucît aceste spaţii sînt prea mari pentru o căutare exhaustivă, euristica este folosită pentru a reduce aria căutărilor. În cazul unor spaţii de căutare mici, ca în jocul de „trei-în-linie", căutarea exhaustivă este preferabilă deoarece este garantată găsirea răspunsului corect.

În general, spaţiul căutării este mare întrucît posibilităţile sînt produse formînd combinaţii ale unor elemente mai simple — mutările individuale în şah, deplasarea din oraş în oraş în problema comis-voiajorului. Această combinare a elementelor conduce la o *explozie combinatorială* a posibilităţilor — o explozie care creşte exponenţial cu numărul elementelor ce sînt combinate. Deoarece posibilităţile sînt constituite din combinaţii de elemente, există o noţiune de distanţă în spaţiu; combinaţiile care au elemente în comun sînt „mai apropiate" decît combinaţiile care nu au. Acesta este motivul pentru care este numit „spaţiu" şi nu doar o mulţime de posibilităţi. Pentru a extinde analogia, ne putem imagina posibilităţile ca fiind amplasate într-un peisaj bidimensional cunoscut uneori ca *peisaj de adecvare*.

Gradul de adecvare sau nota fiecărei soluţii posibile este reprezentată de altitudinea unui punct în peisaj. Dacă posibilităţile asemănătoare au note asemănătoare, atunci punctele apropiate vor avea altitudini similare, astfel încît peisajul va avea văi şi dealuri bine definite. În această analogie, a găsi cea mai bună soluţie este asemenea găsirii vîrfului celui mai înalt deal. Luînd ca exemplu problema comis-voiajorului, ne putem imagina fiecare punct în peisaj ca reprezentînd un anumit traseu de călătorie pentru vînzător. Înălţimea fiecărui punct reprezintă distanţa pe care vînzătorul trebuie să o parcurgă, itinerariile mai eficiente fiind reprezentate de puncte aflate la înălţime mai mare. Cel mai bun traseu va fi pe vîrful celui mai înalt deal.

Unul din cele mai simple moduri de a căuta într-un astfel de spaţiu constă în a compara punctele la întîmplare şi a memora pe cel mai bun găsit. Numărul punctelor ce pot fi explorate în acest mod este în general limitat numai de intervalul de timp disponibil, iar procedeul poate fi aplicat oricărui tip de spaţiu. El este echivalent cu a paraşuta nişte cercetaşi în diferite locuri ale peisajului, cerîndu-le să raporteze la ce înălţime se află. Nu este un mod foarte eficient de a afla vîrful unui deal. Dacă spaţiul este mare, atunci în orice interval rezonabil de timp va fi examinată doar o porţiune infimă din posibilităţi, şi de aceea este puţin probabil ca cel mai bun punct găsit să fie unul din cele mai înalte.

Într-un spaţiu al căutării ca acela din problema comis-voiajorului, unde punctele apropiate au probabil note asemănătoare, de obicei este mai bine să folosim o procedură ce caută un drum călătorind dintr-un punct într-un punct apropiat în spaţiu. La fel cum cea mai bună metodă pentru a găsi un pisc într-un peisaj deluros este de a urca pieptiş, euristica echivalentă este de a

alege cea mai bună dintre soluțiile apropiate aflate în spațiul căutării. În problema comis-voiajorului, de exemplu, calculatorul ar putea modifica cea mai bună soluție cunoscută prin schimbarea ordinii a două din orașele traseului. Dacă această modificare conduce la un itinerariu mai eficient, atunci aceasta este acceptată ca o soluție superioară (un pas în sus); altminteri este respinsă și este încercată o nouă variație. Cu această metodă de căutare se va cutreiera prin spațiu, călătorind întotdeauna în direcție ascendentă, pînă ce se atinge vîrful unui deal. În acel punct, soluția nu poate fi îmbunătățită prin schimbarea vreunei perechi de orașe.

Punctul slab al acestei metode, numită *cățăratul pe deal*, este acela că, deși în felul acesta atingi vîrful unui deal, acesta nu este neapărat cel mai înalt din peisaj. Cățăratul pe deal este o euristică, nu un algoritm. Există alte euristici, asemănătoare cu cățăratul pe deal, care au mai puține șanse de a vă fixa pe vîrful unei coline. De exemplu, puteți repeta de multe ori cățăratul pe deal, pornind din locuri diferite, alese întîmplător (adică puteți cere parașutiștilor să se cațere pe deal). Sau puteți uneori să faceți un pas descendent pentru a evita să fiți fixat. Sînt multe asemenea variații, fiecare cu propriile avantaje și dezavantaje.

Reguli euristice precum cățăratul pe deal funcționează bine în problema comis-voiajorului și produc răspunsuri bune într-un interval de timp scurt. Chiar atunci cînd sînt mii de orașe implicate, de obicei este posibil a găsi o soluție bună pentru problemă pornind de la o presupunere rezonabilă și îmbunătățind-o prin metoda cățăratului pe deal. Și atunci, de ce este dificilă problema comis-voiajorului? Folosind reguli euristice, putem *aproape* întotdeauna obține *aproape* cel mai bun itinerariu. Dar o metodă care funcționează *aproape*

întotdeauna nu este un algoritm. Periodic, mare agitație este produsă de cineva care a „rezolvat“ problema comis-voiajorului; pînă acum, tot ceea ce a făcut de fapt oricare dintre ei este să vină cu o nouă regulă euristică. Nu sînt dificil de imaginat soluții euristice rapide la problema comis-voiajorului; dificultatea este de a găsi un algoritm rapid.

Există multe probleme pentru care nu avem tot timpul nevoie exact de răspunsul corect — probleme pentru care putem accepta o soluție mai puțin decît perfectă. Chiar cînd dorim un răspuns perfect, s-ar putea să nu ni-l permitem. Pentru astfel de probleme, calculatoarele pot produce o evaluare bine studiată și pregătită. Deoarece calculatorul este capabil să examineze un număr enorm de combinații și posibilități, o astfel de evaluare va surprinde deseori programatorul. Cînd un calculator folosește reguli euristice, el este capabil și de surprize, dar și de greșeli — ceea ce îl face puțin mai asemănător unei persoane și mai puțin asemănător unei mașini.

CAPITOLUL 6

Memorie: informații și coduri secrete

Pînă acum, de cele mai multe ori am ignorat limitările impuse unui calculator de mărimea memoriei sale. Un calculator universal idealizat are o memorie infinită, dar memoria unui calculator real este limitată, de obicei de costuri. Cîtă vreme dimensiunea memoriei este corespunzătoare sarcinii respective, sîntem liberi să ignorăm limitarea, dar unele aplicații și unii algoritmi mari consumatori de memorie înmagazinează o cantitate atît de mare de date încît volumul de memorie disponibilă devine de mare importanță. Aplicațiile care manipulează reprezentări ale lumii fizice — precum imagini, sunete sau modele tridimensionale — sînt deseori mari consumatoare de memorie. A ști cîtă memorie este cerută de o aplicație dată este important nu numai pentru a aprecia dacă un calculator este sau nu suficient de mare pentru a manipula informația, ci și pentru a estima timpul necesar prelucrării ei.

Bitul — unitatea de măsură pentru informație — este potrivit atît pentru comunicarea informației, cît și pentru înmagazinarea ei. Într-un sens, comunicarea și stocarea sînt doar două aspecte ale aceluiași lucru: comunicarea trimite un mesaj dintr-un loc în altul; stocarea

„trimite“ un mesaj de la un moment la altul. Cu excepția cazului în care sînteți obișnuit să gîndiți în termeni de spațiu-timp cvadridimensional, această echivalență între mișcare și stocare poate părea ciudată, dar gîndiți-vă la expedierea unei scrisori ca la un mijloc de comunicație care are ambele aspecte. Expedierea unei scrisori către altcineva este un mod de a deplasa informația în spațiu; expedierea unei scrisori către dumneavoastră înșivă este un mod de a stoca informația în timp. La o examinare atentă, orice formă de comunicare poate fi privită ca avînd ambele aspecte — spațial și temporal. Un mod în care calculatoarele electronice înmagazinează informația este să o repună constant în circulație, în echivalentul electronic al scrisorii autoadresate.

Știm că un calculator cu *n* biți de memorie poate păstra pînă la *n* biți de informație, dar cum determinăm cîți biți sînt necesari pentru a reprezenta o anumită bucată de informație ? De exemplu, cîți biți sînt în cuvintele acestei cărți ? Calcularea răspunsului se dovedește a nu fi deosebit de ușoară; de fapt, există mai multe răspunsuri corecte diferite. Reflectînd la această întrebare, sîntem conduși la ideile de comprimare, detectarea și corectarea erorilor, numere aleatoare și coduri secrete.

Numărul de biți necesari pentru a trimite sau stoca o anumită cantitate de date va depinde de modul în care datele sînt codificate. Un mod de a reprezenta un mesaj complex, precum textul unei cărți, constă în a-l reprezenta ca o secvență de părți mai simple: de exemplu, literele cu care se scrie textul. În această reprezentare obișnuită, numărul de biți în mesaj este egal cu numărul de litere din text înmulțit cu numărul de

biți pe caracter. Textul acestei cărți conține cam două sute cincizeci de mii de litere, și calculatorul meu folosește un cod căruia îi sînt necesari opt biți (un octet) pentru a stoca o literă, astfel încît mărimea fișierului pe care calculatorul îl folosește pentru a înmagazina textul este de aproximativ două milioane de biți. Ați putea fi tentați să trageți concluzia că această carte conține cam două milioane de biți de informație, deoarece atîta memorie folosește calculatorul pentru a stoca textul, dar aceasta este doar o măsură a informației — o măsură care depinde de reprezentarea mesajului. Este o măsură folositoare, deoarece vă spune nu numai de cîtă memorie are nevoie calculatorul pentru a stoca informația, ci și de cît timp este nevoie pentru a o prelucra. De exemplu, dacă știu despre calculatorul meu că poate scrie informație pe disc cu viteza de douăzeci de milioane de biți pe secundă și știu că folosește o reprezentare cu două milioane de biți pentru această carte, atunci pot calcula că îi va lua cam o zecime de secundă pentru a înmagazina cartea într-un fișier pe disc.

Folosirea numărului de litere înmulțit cu opt ca o măsură pentru numărul biților în text are neajunsul că depinde de schema de reprezentare folosită de calculator. Un calculator diferit sau un program aplicație diferit, rulînd pe același calculator, pot stoca exact același șir de litere într-un număr diferit de biți. De exemplu, cu opt biți pe caracter este posibilă reprezentarea a două sute cincizeci și șase de caractere diferite, dar textul acestei cărți folosește mai puțin de șaizeci și patru de caractere diferite — cîte douăzeci și șase de litere majuscule și minuscule, cifre și semne de punctuație. Prin urmare, un cod mai eficient ar reprezenta fiecare caracter folosind șase biți de informație (2^6=64) și deci

ar comprima reprezentarea textului la numai un milion şi jumătate de biţi.

Ar fi bine să avem o măsură a informaţiei care să nu depindă de forma de reprezentare. O măsură fundamentală de informaţie ar fi numărul *minim* de biţi necesari pentru reprezentarea textului. Aceasta este uşor de definit, dar nu neapărat uşor de calculat.

COMPRIMARE

Cît de mult putem comprima un text dat fără a pierde informaţie ? Reducerea numărului de biţi pe caracter de la opt la şase este o formă simplă de comprimare. Alte metode de comprimare se bazează pe folosirea regularităţilor din text. De exemplu, în cele mai multe texte în limba engleză, literele T şi E apar cu mult mai des decît literele Q şi Z. Un cod mai eficient ar folosi un şir mai scurt de zerouri şi unuri pentru a reprezenta cele mai frecvente litere. Folosirea unei codificări cu lungime variabilă a caracterelor pentru a realiza o reprezentare mai compactă a fost o stratagemă folosită de primii telegrafişti şi radioamatori. În codul Morse, litera E este reprezentată de un singur punct şi litera T de o singură linie. Literele mai puţin frecvente, precum Q şi Z, sînt reprezentate de şiruri de pînă la patru puncte şi linii. Deoarece un al treilea tip de semnal — un spaţiu — este folosit pentru a marca sfîrşiturile de litere, punctele şi liniile codului Morse nu corespund exact cu zerouri şi unuri, dar principiul este asemănător.

Pentru a obţine un cod cu lungime variabilă folosind zerouri şi unuri, este nevoie să se aleagă cu grijă modelele codului, astfel încît un curent de biţi să poată fi fragmentat neambiguu în caractere. Acest lucru este

posibil atîta vreme cît nici un şir de biţi folosit pentru a reprezenta un caracter nu începe cu un subşir de zerouri şi unuri folosit pentru a reprezenta un alt caracter. De exemplu, toate caracterele obişnuite ar putea fi reprezentate cu patru biţi, începînd cu un unu, în vreme ce caracterele mai puţin obişnuite ar putea avea şapte biţi şi ar putea să înceapă cu un zero. Aceasta ar permite ca un curent de biţi să poată fi împărţit neambiguu în caractere scurte şi lungi. Alegerea unui cod de lungime variabilă care să valorifice cel mai bine frecvenţa relativă a diferitelor litere va avea ca rezultat o comprimare substanţială a textului. În cazul textului acestei cărţi, ar reduce numărul biţilor de la două milioane iniţial la aproximativ un milion, o comprimare de cincizeci de procente.

Orice metodă de comprimare profită de regularităţile din date. Codul pe care tocmai l-am descris profită de regularităţile în frecvenţa de apariţie a caracterelor izolate, dar există alte regularităţi care pot fi exploatate. De exemplu, nu toate perechile de litere adiacente apar cu frecvenţă egală în această carte. Litera Q este aproape întotdeauna urmată de litera U şi litera Z nu este niciodată urmată de litera K. Prin dezvoltarea unui sistem de cod cu lungime variabilă pentru perechi de litere şi nu pentru litere izolate, putem profita de faptul că nu toate combinaţiile de două litere apar cu frecvenţă egală. Codul poate folosi şiruri de biţi mai scurte pentru cele mai obişnuite perechi şi şiruri mai lungi pentru perechi ce apar rar. Dacă folosim această metodă, numărul biţilor necesari pentru a stoca textul acestei cărţi ar putea fi redus cu alte zece procente, cu o medie de circa trei biţi şi jumătate pe literă.

Un cod şi mai eficient ar profita de regularităţile ce apar în şiruri mai lungi de litere. De exemplu, cuvîn-

tul „the" apare de circa trei mii de ori în acest text.* Ar fi avantajos să utilizăm un cod care folosește un șir de biți relativ scurt pentru a reprezenta acest cuvînt în întregime. În mod asemănător, există multe alte cuvinte, cum ar fi „calculator" și „bit", care apar atît de des în acest text special, încît și ele merită a fi codificate.

În șirurile de litere există de asemenea regularități dincolo de aceste regularități statistice. De exemplu, există regularități în gramatică, în structura frazei și în punctuație care permit comprimarea suplimentară a textului. Dar de la un punct vom începe să obținem un randament diminuat. În cele din urmă, comprimarea care folosește cele mai bune metode statistice disponibile va atinge probabil o mărime medie a reprezentării de mai puțin de doi biți pe caracter — aproximativ un sfert din reprezentarea standard cu opt biți pe caracter.

Comprimarea funcționează destul de bine pe text, dar ea funcționează chiar mai bine pe semnale care sînt reprezentări ale lumii reale, precum imaginile și sunetele. Aceste semnale sînt de obicei introduse în calculator printr-un proces cunoscut sub numele de *conversie analogic–numeric.* Astfel de intrări — nivelul intensității unui sunet, să spunem, sau strălucirea luminii — sînt de obicei variabile continue, semnale analogice. De exemplu, un punct sau un *pixel* într-o fotografie alb-negru poate fi alb sau negru sau oricare din infinitatea de nuanțe intermediare de gri. Deoarece calculatorul nu are nici un mod de a reprezenta o infinitate de posibilități, el simplifică semnalul reducînd fiecare pixel la unul dintr-o mulțime finită de niveluri de gri.

---

* Este vorba de textul original, în limba engleză. (*N.t.*)

Tipic, numărul degradeurilor este o putere a lui doi, astfel încît să încapă într-un anumit număr de biţi de memorie. De exemplu, strălucirea unui punct într-o imagine alb-negru poate fi reprezentată pe opt biţi, astfel încît ar putea fi reprezentate două sute cincizeci şi şase de nuanţe de gri. O imagine de calitate ridicată ar fi reprezentată cu un cod pe doisprezece biţi, producînd patru mii nouăzeci şi şase nuanţe de gri. O imagine color ar putea folosi douăzeci şi patru de biţi pe punct — opt biţi pentru intensitatea fiecăreia din cele trei culori fundamentale.

Celălalt parametru determinant pentru calitatea unei imagini fotografice este *rezoluţia* sa — adică numărul de pixeli folosiţi pentru a o reprezenta. O imagine cu rezoluţie înaltă produsă de o matrice de 1000 × 1000 puncte va fi o reprezentare mai precisă decît o imagine cu o rezoluţie 100 × 100. Totuşi, întrucît reprezentarea unei imagini cu rezoluţie înaltă foloseşte un milion de pixeli în loc de zece mii, calculatorul dumneavoastră va avea nevoie de o sută de ori mai multă memorie pentru memorarea imaginii, iar prelucrarea ei va dura de o sută de ori mai mult. Calitatea costă.

Deoarece imaginile cu rezoluţie înaltă pot conţine un număr mare de biţi, deseori este de dorit să fie comprimate pentru a reduce costul depozitării şi transmiterii. Acest lucru este adevărat îndeosebi pentru imagini în mişcare, care de obicei conţin de la douăzeci şi patru pînă la o sută de cadre pe secundă. Din fericire, imaginile sînt relativ uşor de comprimat deoarece există un grad înalt de regularitate într-o imagine. În cele mai multe fotografii, intensitatea şi culoarea unui pixel sînt deseori identice cu intensitatea şi culoarea pixelilor vecini. Doi pixeli reprezentînd părţi adiacente ale aceluiaşi

obraz în imaginea unei fețe, să spunem, vor fi probabil foarte asemănători în strălucire și culoare. Cei mai mulți algoritmi pentru comprimarea imaginilor profită de această similitudine. Un algoritm pentru comprimarea imaginilor poate reprezenta zone de strălucire și culoare uniforme folosind doar cîteva numere pentru a reprezenta culoarea și mărimea ariei zonei. Alte metode pentru comprimarea imaginilor exploatează forme mai complexe de regularitate: de exemplu, texturi asemănătoare în diferite părți ale imaginii. Pentru imagini în mișcare, cum ar fi transmisii de televiziune, metodele de comprimare exploatează în general similitudinea secvențelor de cadre. Folosind astfel de tehnici, se poate adeseori comprima reprezentarea unei fotografii de zece ori și a unei imagini în mișcare de o sută de ori. Metode de comprimare asemănătoare pot fi aplicate sunetelor.

Aceste metode de comprimare conduc la o noțiune referitoare la cantitatea de informație conținută într-o imagine, noțiune ce contrazice intuiția. Dacă numărul minim de biți necesari pentru a reprezenta imaginea este luat ca o măsură pentru cantitatea de informație din imagine, atunci o imagine care este ușor de comprimat va avea mai puțină informație. O imagine a unei fețe, de exemplu, va avea mai puțină informație decît imaginea unei grămezi de pietricele de pe plajă, deoarece este mai probabil ca pixelii adiacenți în imaginea feței să fie asemănători. Pentru pietricele este necesară stocarea și transmiterea a mai multă informație, chiar dacă un observator uman ar putea considera că fotografia feței conține mult mai multă informație. După această măsură, fotografia cu cea mai multă informație ar fi o ilustrație de pixeli complet aleatori, precum cea

statică de pe ecranul unui televizor defect. Dacă punctele din imagine nu sînt corelate cu vecinii lor, nu există nici o regularitate pentru a comprima. Astfel de fotografii ne par complet fără de înțeles — și ar putea fi într-adevăr nesemnificative — dar ele necesită cantitatea cea mai mare de informație pentru a fi reprezentate de un calculator.

Măsura informației dată de reprezentarea minimală nu corespunde bine intuiției noastre despre conținutul informațional deoarece calculatorul nu distinge informația semnificativă de cea fără sens. Calculatorul trebuie să reprezinte culoarea fiecărui pixel, sau poziția fiecărei pietricele pe plajă, chiar dacă aceste detalii ar putea să nu fie importante pentru privitor. Este o artă subtilă a decide care informație este semnificativă și care nu; depinde cum este folosită o imagine și cine o folosește. Poziția unei pete minuscule într-o radiografie poate fi irelevantă pentru un ochi neantrenat, dar foarte semnificativă pentru un medic. Un artist mare precum Picasso poate fi capabil să „comprime“ imaginea unei scene complexe în cîteva linii simple, dar făcînd acesta el execută un raționament complex, hotărînd care aspecte ale imaginii comunică cel mai mult înțeles (*cf.* Figura 23).

Dacă un calculator ar comprima o imagine stocînd doar informația semnificativă, atunci numărul biților în reprezentare ar corespunde mai îndeaproape cu noțiunea noastră intuitivă despre informația conținută în imagine. De exemplu, calculatorul ar putea reprezenta tabloul de pixeli aleatori indicînd că această imagine nu are nici o regularitate și nici o informație semnificativă. Cînd i s-ar cere să reconstruiască această imagine, el ar putea pur și simplu să genereze un alt tablou de pixeli aleatori. Detaliile — ce nuanță exactă avea fiecare

FIGURA 23 Schiță de Picasso

pixel — ar fi diferite în imaginea originală și în cea reconstruită, dar aceste diferențe ar fi nesemnificative pentru ochiul uman.

Mulți algoritmi pentru comprimarea imaginii și sunetului elimină anumite informații nesemnificative cu scopul de a reduce mărimea reprezentării. Acești algoritmi de comprimare, așa-numita *comprimare prin omisiune*, presupun că un anumit nivel de detaliere în imagine sau sunet va fi ignorat de ochi sau ureche. Metodele de comprimare prin omisiune sînt folosite în general atunci cînd este cunoscut faptul că informația decomprimată va folosi unui scop anume. De exemplu, dacă un anumit detaliu apare numai într-un singur cadru al unui film, el poate fi eliminat fără teamă, deoarece el va trece neobservat.

Există o altă formă importantă de reprezentare a imaginii, care este capabilă să realizeze un grad de comprimare chiar mai mare decît metodele tocmai descrise. Dacă se cunoaşte procesul care generează imaginea originală, atunci ar putea fi mai eficient să se păstreze o înregistrare a acelui proces decît o înregistrare a imaginii înseşi. De exemplu, dacă imaginea a fost un desen creat de o persoană care a desenat o serie de linii, atunci desenul poate fi reprezentat stocînd o listă a liniilor — o schemă de reprezentare folosită deseori de calculatoare pentru a face simple desenele constînd din linii.

Ideea de a reprezenta un lucru stocînd procedura sau programul care l-a generat este aplicabilă şi altor tipuri de date, cum ar fi sunetele. În ceea ce priveşte sunetele, tehnica ar putea părea nu mai profundă decît înregistrarea unei piese muzicale prin scrierea partiturii, dar într-un calculator partitura este capabilă să reprezinte fiecare detaliu necesar pentru a reproduce originalul — acordarea instrumentelor, maniera de a mînui arcuşul, chiar şi starea sufletească a orchestrei. Dacă un obiect poate fi generat de un calculator, atunci prin definiţie există o procedură precisă pentru generarea sa, astfel încît o descriere a acelei proceduri va servi ca o reprezentare a obiectului.

Acest fapt conduce la o altă măsură pentru informaţie: *cantitatea de informaţie într-un eşantion de biţi este egală cu lungimea celui mai scurt program de calculator capabil să genereze acei biţi.* Această definiţie a informaţiei este valabilă dacă eşantionul de biţi reprezintă în definitiv o imagine, un sunet, un text, un număr sau orice altceva. Definiţia este interesantă deoarece permite orice tip de regularitate în eşantion. În particular, ea înglobează toate metodele de comprimare descrise mai sus. (Poate părea că o astfel de definiţie

depinde de detaliile limbajului maşină al calculatorului, dar să ne reamintim că orice calculator poate simula oricare altul, astfel că măsura va varia de la un calculator la altul doar printr-o porţiune mică de cod necesar pentru a executa simularea.)

De îndată ce un şir de informaţii este comprimat atît cît e posibil, el nu va mai prezenta nici o regularitate. Acest lucru este adevărat întrucît orice regularitate va fi o ocazie favorabilă pentru o comprimare suplimentară. Şirul de zerouri şi unuri reprezentînd un text comprimat optimal ar arăta complet aleatoriu, precum înregistrarea aruncărilor unei monede. De fapt, mulţi matematicieni folosesc această proprietate de incompresabilitate ca o definiţie a aleatoriului — o definiţie satisfăcător de simplă, dar care adeseori nu este foarte folositoare în practică, întrucît este foarte dificil să spui dacă o succesiune de biţi dată este sau nu aleatoare în acest sens. Cînd recunoaştem orice regularitate, este uşor de decis că un şir poate fi comprimat — dar nu putem dovedi că şirul *nu poate* fi comprimat dacă nu vedem nici un model. Şirurile de numere pseudoaleatoare descrise în capitolul patru sînt un bun exemplu de şiruri ce par aleatoare, dar care se bazează pe un model. Conform definiţiei de mai sus a aleatoriului, ele sînt în mare măsură nealeatoare, deoarece un şir foarte lung poate fi descris pe scurt descriind pur şi simplu algoritmul care îl produce — în acest caz, simularea roţii de ruletă.

CRIPTARE

Acele şiruri care par aleatorii, dar care se bazează pe un model ascuns, pot fi folosite pentru a crea coduri

pentru *criptarea* datelor. Să ne imaginăm, de exemplu, că doresc să transmit unui prieten al meu un mesaj secret. Dacă amîndoi avem o copie a aceluiași generator de numere aleatoare, astfel că putem genera același șir de numere pseudoaleatoare, am putea folosi acest șir pentru a ascunde conținutul mesajului de oricine l-ar intercepta. Să spunem că mesajul de transmis este un curent de biți reprezentînd caractere, folosind reprezentarea standard de opt biți pe caracter. Probabil că această reprezentare standardizată ar putea fi interpretată și înțeleasă de orice persoană ce ascultă pe ascuns; este ceea ce criptografii numesc *textul în clar* al mesajului. Pentru a cripta mesajul, împerechem fiecare bit din textul în clar cu bitul corespunzător din curentul de biți pseudoaleatori. Dacă bitul pseudoaleator este unu, atunci inversăm bitul corespunzător din textul în clar. Dacă bitul pseudoaleator este zero, lăsăm neschimbat bitul corespunzător din textul în clar. Astfel se vor inversa aproximativ jumătate din biții textului în clar, dar cel ce ne ascultă pe ascuns nu va ști care jumătate. Dacă cel ce ne ascultă pe ascuns nu cunoaște șirul pseudoaleator, secvența de zerouri și unuri va fi complet lipsită de sens. Pe de altă parte, prietenul meu destinatar știe cum să genereze exact același șir aleator, care poate fi folosit pentru a inversa din nou biții inversați, iar prin aceasta să reconstituie (să decripteze) mesajul. Această metodă, sau alta foarte asemănătoare ei, stă la baza celor mai multe scheme de criptare.

Criptarea unui mesaj este analoagă cu trimiterea lui într-o cutie închisă care poate fi deschisă doar cu o cheie specială. În metoda de criptare tocmai descrisă, cheia este generatorul de numere aleatoare. Oricine are cheia este capabil să execute conversia. În exemplul de mai sus, aceeași cheie este folosită pentru criptare

și decriptare, dar este de asemenea posibil să construim coduri care folosesc chei diferite pentru criptare și decriptare. Într-o schemă de criptare publică, cheile pentru criptare și decriptare sînt diferite, iar cineva care ne ascultă pe ascuns și cunoaște cheia de criptare nu va cunoaște prin aceasta cheia necesară pentru decriptare. Această metodă de transmitere a mesajelor este extrem de folositoare. De exemplu, dacă eu doresc să primesc un mesaj criptat, pot face publică descrierea cheii necesare pentru a codifica mesaje adresate mie. Oricine va fi capabil să îmi trimită un mesaj secret, indiferent dacă este sau nu o cunoștință a mea. Deoarece cheia publică spune expeditorului doar cum să cripteze mesajul, nu cum să îl decripteze, ceilalți nu vor fi capabili să descifreze mesajul codificat. Doar cheia mea privată, pe care o țin secretă, permite convertirea unui mesaj criptat înapoi în text în clar. Aceasta este numită *criptare cu cheie publică*. Criptarea cu cheie publică rezolvă o problemă practică importantă: de exemplu, multe firme care acceptă numere de cărți de credit transmise prin rețeaua internet fac cunoscută cheia lor publică, astfel încît clienții pot cripta numerele cărților lor de credit fără teama că ele vor fi interceptate și citite.

Schema de criptare cu cheie publică este de asemenea folositoare în sens invers, pentru autentificarea mesajelor. În acest caz eu fac publică cheia pentru decriptarea unui mesaj, dar țin secretă cheia mea pentru criptare. De cîte ori doresc să trimit un mesaj pe care doresc să îl „semnez“ ca provenind în mod verificabil de la mine, codific mesajul cu cheia mea privată. Orice primitor al mesajului poate folosi cheia publică pentru decriptarea mesajului. Ei vor ști că a fost într-adevăr de la mine, deoarece doar cineva care cunoaște cheia mea privată ar fi putut codifica un astfel de mesaj.

DETECTAREA ERORILOR

Pe lîngă comprimare și securitate, codificarea și decodificarea biților are multe alte aplicații. De exemplu, sînt situații în care reprezentăm un mesaj folosind mai mulți biți decît ar fi necesari, pentru a reduce șansa de eroare. Codurile care folosesc o anume formă de redundanță pentru a detecta erorile de transmisie — de exemplu, un zero a fost recepționat ca un unu — sînt numite *coduri detectoare de erori.* Alte coduri, numite *coduri corectoare de erori*, conțin suficientă informație redundantă pentru a detecta, precum și pentru a corecta astfel de erori.

O formă evidentă de redundanță este trimiterea unui mesaj de mai multe ori. Trimiterea unui mesaj de două ori permite detectarea erorilor. Dacă sînt transmise două cópii ale aceluiași mesaj, dar sînt recepționate mesaje ușor diferite, atunci trebuie să fi apărut anumite erori în transmitere. Un cod *corector* de erori simplu ar fi repetarea mesajului de trei ori. Presupunînd că un singur mesaj a fost alterat, destinatarul ar reconstitui mesajul corect alegînd cele două cópii care ar fi identice.

Din fericire există coduri detectoare de erori și corectoare de erori care obțin rezultate cu mult mai puțină redundanță. O schemă folosită în mod curent pentru detectarea erorilor este un *cod de paritate.* Această schemă poate detecta o eroare de un singur bit într-un mesaj de lungime arbitrară, prin adăugarea unui bit redundant. Ca exemplu concret de cod de paritate, să considerăm codul de opt biți utilizat deseori pentru transmiterea caracterelor prin linii de comunicație cu zgomote. Bitul al optulea, numit *bitul de paritate*, este unu dacă și numai dacă numărul de unuri în ceilalți șapte biți este par. Aceasta înseamnă că numărul de

unuri din șirul de opt biți ar trebui să fie totdeauna impar. Dacă zgomotul în linia de transmisie a datelor provoacă recepționarea unui unu ca un zero sau invers, atunci mesajul de opt biți care este recepționat va avea un număr par de unuri. Destinatarul va fi prin urmare capabil să descopere că a fost o eroare. Scheme de paritate asemănătoare sînt folosite pentru detectarea erorilor în sistemele de memorie ale calculatoarelor. Un bit de paritate poate fi folosit pentru detectarea unei erori într-un mesaj constînd din orice număr de biți. O limitare a acestui cod de paritate simplu este faptul că este bun doar la detectarea unei singure erori. Un mesaj care are doi biți inversați va avea paritatea corectă, chiar dacă datele sînt incorecte.

Detectarea erorilor multiple este posibilă prin folosirea mai multor biți de paritate. Este de asemenea posibil să dai destinatarului informație suficientă nu numai pentru a detecta o eroare, ci și pentru a o corecta; adică destinatarul este capabil să reconstituie mesajul originar chiar dacă a existat o eroare. Un exemplu al unui asemenea cod este codul de paritate bidimensional ilustrat în Figura 24.

Acest cod conține nouă biți pentru informația mesajului și șase biți de paritate. Biții mesajului sînt aranjați în trei rînduri a cîte trei biți fiecare. Există un bit de paritate pentru fiecare linie (orizontală) și un bit pentru fiecare coloană (verticală). O eroare de un singur bit într-un mesaj va provoca detectarea a două erori de paritate, una într-o linie și cealaltă într-o coloană. Destinatarul mesajului va ști atunci că bitul de la intersecția coloanei eronate cu linia eronată este incorect și trebuie inversat. Pe de altă parte, dacă există o eroare în transmiterea unuia din biții de paritate, atunci fie o linie, fie o coloană va prezenta paritate

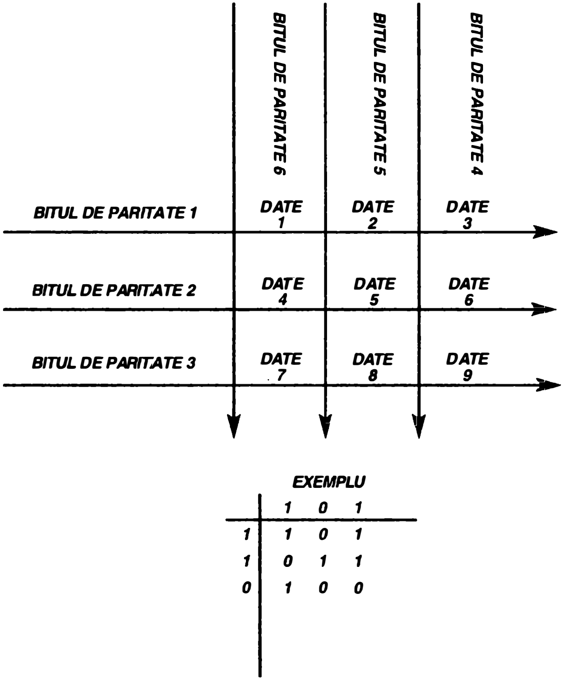


FIGURA 24 Un cod corector de erori
cu nouă biți de date și șase biți de control

incorectă, dar nu ambele. Biții sînt desenați într-un model bidimensional pentru a vă ajuta să vizualizați structura codului, dar ei pot fi transmiși în orice ordine. Astfel de coduri corectoare de erori sînt deseori folosite pentru a proteja fiecare cuvînt din memoria unui calculator mare. Folosind tehnici asemănătoare, pot fi construite multe alte coduri, coduri care vor detecta și/sau corecta un număr diferit de erori de diverse tipuri.

Codurile corectoare de erori sînt capabile să semnaleze erori care apar la transmiterea și stocarea infor-

maţiei, dar erorile de calcul înseşi pot fi detectate ? Se dovedeşte că este de asemenea posibil să construim blocuri logice care produc răspunsuri corecte, chiar dacă unele dintre blocurile din care sînt construite funcţionează incorect. Din nou, instrumentul fundamental este o formă de redundanţă. Un mod de a construi un bloc logic care tolerează greşelile este copierea fiecărui bloc logic de trei ori. Un bloc ce implementează funcţia majoritate, ca acela din Figura 12 din capitolul doi, poate fi folosit pentru a combina răspunsurile celor trei cópii. Dacă una din cópii conţine o greşeală, ea va fi pusă în minoritate de celelalte. Această metodă simplă protejează împotriva unei singure erori (cu excepţia uneia în însuşi blocul majoritate).

Fiind dată o mulţime bine definită de erori posibile — cum ar fi ruperea firelor, blocarea comutatoarelor şi schimbarea zerourilor în unuri — este posibilă construirea unui dispozitiv de calcul oricît de sigur din componente oricît de nesigure. Această sarcină necesită pur şi simplu să folosim suficientă logică redundantă într-o formă sistematică. De exemplu, dacă aţi putut construi un nou tip de comutator — să spunem un comutator molecular — care este extrem de rapid sau extrem de ieftin, dar care dă erori într-o cincime din situaţii, tot aţi putea să îl folosiţi pentru a construi un calculator care produce răspunsuri corecte în 99,9999% din situaţii, asigurînd redundanţa adecvată în circuite.

Acest lucru înseamnă că puteţi construi un calculator oricît de sigur ? Nu e întru totul exact. Deşi se poate construi un calculator astfel încît să se elimine un anumit tip de eroare, pot să apară erori de tipuri neprevăzute, ceea ce provoacă erori corelate în modulul redundant. De exemplu, arderea unui modul poate provoca

supraîncălzirea altui modul, sau un anumit fel de pulsație magnetică poate chiar face ca toate modulele să greșească simultan. Inginerii sînt capabili să proiecteze un bloc logic care poate trata orice tip de eroare pe care îl pot imagina, dar istoria tehnologiei arată că imaginația noastră nu este întotdeauna suficientă. Cele mai dramatice eșecuri sînt de obicei surprizele.

Un al doilea motiv pentru a nu spera să obținem un calculator perfect este faptul că cele mai multe erori ale calculatoarelor nu sînt provocate de funcționarea incorectă a logicii. Ele provin din erori în proiect — de obicei în proiectul de software. Calculatoarele programate, inclusiv software-ul lor, sînt de departe cele mai complexe sisteme proiectate vreodată de ființele umane. Numărul componentelor ce interacționează într-un calculator este cu cîteva ordine de mărime mai mare decît numărul componentelor celui mai complex avion. Metodele moderne de inginerie nu se ridică de obicei la nivelul cerut pentru proiectarea unor obiecte de asemenea complexitate. Un calculator modern poate avea literalmente milioane de operații logice desfășurîndu-se simultan și sînt imposibil de anticipat consecințele fiecărei combinații posibile de evenimente. Metodele de abstractizare funcțională descrise în capitolele precedente ajută la ținerea sub control a interacțiunilor, dar aceste abstractizări depind de măsura în care totul interacționează conform așteptărilor. Cînd apar interacțiuni neanticipate (și ele apar), presupunerea pe care se bazează abstractizarea eșuează, iar consecințele pot fi catastrofale. Într-un sens practic, comportarea unui sistem de calcul mare este uneori imprevizibilă, chiar dacă nu apar eșecuri — și acesta este motivul esențial pentru care este imposibil să proiectezi un calculator perfect sigur.

CAPITOLUL 7

Viteză: calculatoare paralele

Pe lîngă diferențe în cantitatea de memorie, un tip de calculator universal poate prezenta diferențe față de un altul în privința vitezei cu care sînt executate operațiile sale. Viteza unui calculator este în general determinată de timpul necesar pentru a muta datele în și din memorie.

Calculatoarele despre care am vorbit pînă acum sînt de tipul calculatoare seriale — la un moment dat ele operează cu un singur cuvînt de date. Motivul pentru care calculatoarele obișnuite funcționează în acest mod ține în mare măsură de istorie. La sfîrșitul anilor 1940 și la începutul anilor 1950, cînd calculatoarele se dezvoltau, elementele comutatoare (relee și tuburi electronice cu vid) erau scumpe, dar relativ rapide. Elementele de memorie (circuite de întîrziere cu mercur, tambure magnetice) erau relativ ieftine și lente. Ele erau potrivite îndeosebi pentru a produce fluxuri secvențiale de date. Calculatoarele erau proiectate pentru a ține ocupate în mod folositor comutatoarele scumpe din procesor pe cît de mult posibil, fără a solicita prea mult viteza memoriei. Aceste calculatoare din prima generație erau de mărimea unei camere, cu procesorul scump

într-o parte și cu memoria înceată de cealaltă parte. Un firicel de date curgea între ele.

Pe măsură ce tehnologia calculatoarelor se îmbunătățea, programele au devenit din ce în ce mai complexe și costisitoare, și a devenit din ce în ce mai dificil să pregătești programatori, astfel că, pentru a economisi investiția în programe și pregătire, structura de bază a calculatoarelor a rămas neschimbată. Existau puține motive pentru a regîndi proiectul în două părți, deoarece tehnologia progresa într-un ritm atît de rapid încît era ușor să construiești mașini mai rapide și mai ieftine numai prin re-crearea aceluiași tip de calculator într-o tehnologie nouă.

Viteza calculatoarelor tindea să se dubleze la fiecare doi ani. Tuburile electronice cu vid au fost înlocuite de tranzistori și, în cele din urmă, de circuitele integrate. Memoriile cu circuit de întîrziere au fost înlocuite de memorii cu miez magnetic, și apoi tot de circuitele integrate. Mașinile de mărimea unei camere s-au contractat la o pastilă de siliciu de mărimea unei unghii. Prin toate aceste tehnologii în schimbare, proiectul simplu al unui procesor conectat la o memorie a rămas același. Dacă priviți la microscop un calculator modern pe o pastilă de siliciu, puteți încă vedea vestigiile camerei pline cu lămpi: o zonă a pastilei este rezervată procesării, alta memoriei. Acest lucru este adevărat în ciuda faptului că părțile de procesare și de memorare ale calculatorului sînt produse acum prin aceleași metode, adeseori pe o aceeași bucată de siliciu. Modelele de activitate sînt încă optimizate pentru proiectul de început, cu două părți. Secțiunea din pastila de siliciu care implementează procesarea este ținută foarte ocupată; partea care implementează memoria

încă lasă să se scurgă doar cîte un cuvînt de date la un moment dat.

Fluxul de date între procesor și memorie este locul îngust al unui calculator serial. Problema esențială este că memoria este proiectată să aibă acces la o singură locație de memorie la fiecare ciclu de instrucțiuni. Cîtă vreme rămînem credincioși acestui proiect fundamental, singurul mod de a crește viteza calculatorului este să reducem durata ciclului. Vreme de mai mulți ani, durata ciclului calculatoarelor a putut fi redusă mărind viteza comutatoarelor: comutatoare mai rapide însemnau calculatoare mai rapide. Acum, aceasta nu mai e o strategie eficientă. Viteza calculatoarelor mari de astăzi este limitată în principal de timpul necesar pentru propagarea informației de-a lungul circuitelor, iar aceasta este limitată la rîndul său de viteza finită a luminii. Lumina călătorește cam treizeci de centimetri într-o nanosecundă (o miliardime de secundă). Durata unui ciclu la cele mai rapide calculatoare de astăzi este cam o nanosecundă, și nu este o coincidență faptul că procesoarele lor nu depășesc treizeci de centimetri. Ne apropiem de limitele pînă la care putem spori viteza calculatorului fără a-i schimba proiectul de bază.

PARALELISM

Pentru a funcționa mai repede, calculatoarele de astăzi trebuie să execute mai mult de o operație deodată. Putem realiza aceasta prin divizarea memoriei calculatorului într-un număr mare de memorii mici și dînd fiecăreia propriul său procesor. O astfel de mașină este numită *calculator paralel.* Calculatoarele paralele sînt practice datorită costului scăzut și mărimii reduse a

microprocesoarelor. Putem construi un calculator paralel legînd împreună zeci, sute sau chiar mii de asemenea calculatoare mai mici. Cele mai rapide calculatoare din lume sînt calculatoarele *masiv paralele,* care folosesc mii sau chiar zeci de mii de procesoare.

După cum am descris mai devreme, calculatoarele sînt constituite dintr-o ierarhie de blocuri constructive, cu fiecare nivel al ierarhiei servind ca treaptă pentru nivelul de deasupra. Calculatoarele paralele sînt în mod evident următorul nivel în această schemă, calculatoarele înseși devenind blocuri constructive. O astfel de construcție poate fi numită fie calculator paralel, fie o *rețea de calculatoare.* Distincția dintre cele două este întru cîtva arbitrară și are mai mult de-a face cu modul în care este folosit sistemul decît cu modul în care funcționează. De obicei, un calculator paralel se află într-un singur loc, în vreme ce o rețea este răspîndită geografic, dar există excepții la ambele aceste reguli. În general, dacă un grup de calculatoare conectate sînt folosite în mod coordonat, le numim calculator paralel. Dacă însă calculatoarele sînt folosite întru cîtva independent, numim calculatoarele conectate o rețea de calculatoare.

Punerea laolaltă a unui mare număr de calculatoare pentru a obține o viteză mai mare pare un pas evident, dar mulți ani consensul între informaticieni a fost că ar funcționa doar pentru puține aplicații. La începutul carierei mele mi-am petrecut o mare parte din timp discutînd în contradictoriu cu oamenii care credeau că era nepractică sau chiar imposibilă construirea și programarea calculatoarelor masiv paralele de uz general. Acest scepticism răspîndit era bazat pe două neînțelegeri — una referitoare la cît de complex ar fi un asemenea sistem și cealaltă referitoare la felul în care

ar funcționa împreună componentele unui asemenea sistem.

Oamenii de știință tindeau să supraestimeze complexitatea calculatoarelor paralele deoarece ei subestimau — sau cel puțin subapreciau — ritmul de perfecționare a tehnologiei de fabricație în microelectronică. Nu atît pentru că ei nu cunoșteau această tendință, ci pentru că ritmul de schimbare tehnologică era fără precedent în istorie; era astfel foarte dificil ca așteptările și intuițiile să fie la curent cu schimbarea. Îmi amintesc o comunicare prezentată la o conferință despre calculatoare ținută la hotelul Hilton din New York pe la mijlocul anilor 1970 și în care semnalam că tendințele actuale indicau că în Statele Unite vor fi în curînd mai multe microprocesoare decît locuitori. Aceasta a fost considerată o extrapolare exagerată în acel timp. Deși microprocesoarele fuseseră deja produse, viziunea obișnuită asupra unui calculator era încă aceea a unei mulțimi mari de dulapuri de mărimea unui frigider și cu lumini clipitoare. În partea de la sfîrșitul expunerii, rezervată întrebărilor, unul din ascultătorii mei prost-dispuși a întrebat cu o voce încărcată de sarcasm: „Ce crezi exact că vor face oamenii cu toate aceste calculatoare? Că doar nu ai nevoie de un calculator în fiecare clanță de ușă!“ Publicul a izbucnit în rîs și am găsit cu greu un răspuns, dar este un fapt că în același hotel astăzi fiecare clanță conține un microprocesor care controlează încuietoarea.

Un alt motiv pentru care oamenii erau sceptici în privința calculatoarelor paralele era mai subtil și, de asemenea, mai greu de combătut. Era percepută ineficiența în fragmentarea unui calcul în multe părți simultane. Această problemă continuă să limiteze utilizarea calculatoarelor paralele astăzi, dar nu este o limitare atît de

serioasă pe cît s-a considerat cîndva a fi. O parte din motivul pentru supraestimarea dificultății s-a legat de o serie de experiențe înșelătoare cu primele mașini paralele. Primele calculatoare numerice paralele au fost construite în anii 1960 prin conectarea a două sau trei calculatoare seriale mari. În cele mai multe cazuri, mai multe unități de procesare împărțeau o singură memorie, astfel încît fiecare dintre procesoare putea avea acces la aceleași date. Aceste calculatoare paralele din prima generație erau de obicei programate pentru a da fiecărui procesor o sarcină diferită: de exemplu, într-o aplicație de baze de date, un procesor ar regăsi înregistrarea, alt procesor ar întocmi tabele statistice, iar al treilea ar tipări rezultatele. Procesoarele erau folosite asemănător cu muncitori diferiți pe o linie de asamblare, fiecare executînd o fază a calculului.

Existau cîteva ineficiențe inerente în această schemă, toate părînd a crește o dată cu numărul procesoarelor. O ineficiență era că sarcina trebuia fragmentată în faze mai mult sau mai puțin independente. În timp ce era adeseori posibil să divizezi sarcina în două sau trei faze, era dificil de văzut cum să împarți o sarcină în zece sau o sută de faze. Un detractor al prelucrării paralele a explicat astfel unui reporter de la un ziar: „Vedeți, doi reporteri ar putea fi în stare să scrie mai repede un articol de ziar dacă unul din ei adună știrile în timp ce celălalt scrie articolul, dar o sută de reporteri lucrînd împreună la un articol probabil nu vor fi în stare să îl scrie deloc." Astfel de argumente erau destul de convingătoare.

Altă ineficiență provenea din accesul partajat la o singură memorie. O memorie tipică putea regăsi la fiecare moment doar un cuvînt dintr-o zonă dată a memoriei și evident că această limitare pentru viteza de acces

crea în sistem un loc îngust care limita performanțele sale. Dacă mai multe procesoare ar fi adăugate unui sistem deja limitat de viteza sa de extragere a datelor, procesoarele vor sfîrși cu toate cheltuind mai mult timp așteptînd date și eficiența sistemului ar descrește.

Mai mult, procesoarele trebuiau să fie atente să nu creeze o incompatibilitate prin modificarea datelor pe care le căuta un partener. Să considerăm, de exemplu, un sistem de rezervare a biletelor de avion. Dacă un procesor lucrează la problema rezervării locurilor, el caută să vadă dacă un loc e liber și apoi rezervă locul dacă acesta este liber. Dacă două procesoare vînd simultan locuri pentru doi pasageri diferiți, există un neajuns potențial: ar putea să observe amîndouă că un același loc e liber și să decidă rezervarea sa înainte ca fiecare să aibă șansa de a-l semnala ca ocupat. Pentru a evita această întîmplare nefericită, un procesor trebuia să treacă printr-o secvență complicată care bloca accesul celorlalte procesoare la date atîta timp cît acel procesor examina un cuvînt de date. Aceasta agrava suplimentar ineficiențele ce însoțeau competiția pentru memoria sistemului și în cel mai rău caz reducea viteza unui sistem multiprocesor la viteza unui singur procesor — și chiar la mai puțin decît viteza unui singur procesor. Cum am observat, aceste ineficiențe se înrăutățeau cu creșterea numărului de procesoare.

Cea din urmă sursă a ineficienței apărea a fi chiar mai importantă: dificultatea de a echilibra sarcinile atribuite diferitelor procesoare. Pentru a reveni la analogia cu linia de asamblare: putem vedea că viteza de calcul va fi determinată de viteza celui mai lent pas. Dacă există doar o singură operație lentă, viteza de calcul este fixată de acea unică operație. Nu este nerezonabil să ne așteptăm ca și în acest caz eficiența sistemului

să descrească o dată cu creșterea numărului de procesoare.

Cea mai bună formulare a acestor probleme legate de ineficiență este cunoscută ca legea lui Amdahl, după Gene Amdahl, proiectantul de calculatoare care a propus-o în anii 1960. Raționamentul lui Amdahl era cam așa: totdeauna va fi o parte a calculului care este inerent secvențială — treabă ce poate fi făcută la un moment dat numai de un singur procesor. Chiar dacă numai zece procente din calcul este inerent secvențial, indiferent de cît de mult accelerezi cele nouăzeci de procente rămase, calculul în întregul său nu va putea fi accelerat de mai mult de zece ori. Procesoarele lucrînd la cele nouăzeci de procente ce pot fi executate în paralel vor ajunge să aștepe ca acel unic procesor să termine cele zece procente secvențiale ale sarcinii. Acest raționament sugerează că un calculator paralel cu o mie de procesoare ar fi extrem de ineficient, din moment ce va fi doar de zece ori mai rapid decît un singur procesor. Atunci cînd am încercat să găsesc finanțare pentru a construi primul meu calculator paralel — un calculator masiv paralel cu șaizeci și patru de mii de procesoare — prima întrebare care mi se punea de obicei la sfîrșitul uneia din tiradele mele era: „Nu ai auzit niciodată de legea lui Amdahl ?“

Bineînțeles, *auzisem* de legea lui Amdahl și, ca să vă spun adevărul, nu vedeam nimic greșit în raționamentul din spatele său. Și totuși, deși nu puteam să o dovedesc, eram sigur că legea lui Amdahl nu se aplica problemelor pe care încercam eu să le rezolv. Motivul pentru care eram atît de încrezător este acela că problemele la care lucram fuseseră deja rezolvate pe un procesor masiv paralel — creierul uman. Eram student la Labo-

ratorul de Inteligență Artificială de la Institutul de Tehnologie din Massachusetts și doream să construiesc o mașină care putea gîndi.

În 1974, cînd am vizitat prima oară Laboratorul de Inteligență Artificială de la Institutul de Tehnologie din Massachusetts ca student în anul întîi, domeniul inteligenței artificiale (așa cum începea să fie cunoscut) era într-o stare de creștere explozivă. Primele programe care puteau urma instrucțiuni simple scrise în limba engleză obișnuită fuseseră dezvoltate, iar apariția unui calculator care să înțeleagă vorbirea umană părea iminentă. Calculatoarele excelau la jocuri, precum șahul, care fuseseră considerate prea complicate pentru ele cu doar cîțiva ani în urmă. Sistemele de vedere artificială recunoșteau obiecte simple, precum desene liniare și grămezi de blocuri. Calculatoarele treceau chiar examene de analiză matematică și rezolvau probleme simple de analogie luate din testele pentru stabilirea coeficientului de inteligență. Putea fi inteligența unei mașini de uz general atît de departe de toate acestea ?

Dar cîțiva ani mai tîrziu, cînd am devenit membru al laboratorului ca student la specializare, problemele păreau mai complicate. Demonstrațiile simple s-au dovedit a fi doar atît. Deși fuseseră inventate multe principii noi și metode puternice, aplicarea lor la probleme mai mari, mai complicate, părea a nu merge. Cel puțin o parte a dificultăților se datorau limitărilor vitezei calculatoarelor. Cercetătorii în inteligență artificială au constatat că e infructuos să extindă experimentele lor la cazuri în care apar mai multe date, deoarece calculatoarele erau deja lente, iar adăugarea mai multor date doar le făcea mai încete. Era frustrant, de exemplu, să încerci să faci o mașină să recunoască o grămadă de

obiecte cînd recunoașterea unui singur obiect necesita ore de calcul.

Calculatoarele erau lente deoarece erau seriale; ele puteau face doar un singur lucru la un moment dat. Spre deosebire de un calculator care trebuie să privească o imagine punct cu punct, creierul percepe instantaneu o întreagă imagine și simultan poate asocia ceea ce vede cu fiecare imagine pe care o cunoaște. Din acest motiv, o ființă umană este cu mult mai rapidă decît un calculator la recunoașterea obiectelor, chiar dacă neuronii din sistemul vizual uman sînt mult mai lenți decît tranzistorii din calculator. Această diferență în proiect m-a inspirat, cum a inspirat pe mulți alții, să caut moduri de a proiecta calculatoare masiv paralele — calculatoare care puteau executa milioane de operații simultan și puteau exploata paralelismul mult mai asemănător modului în care o face creierul. Întrucît știam despre creier că era capabil să obțină performanțe superioare cu componente lente, știam și că legea lui Amdahl nu este aplicabilă întotdeauna.

Acum înțeleg eroarea din raționamentul lui Amdahl. Ea constă în presupunerea că o parte fixată din calcul, chiar numai zece procente, *trebuie* să fie secvențială. Această estimare pare plauzibilă, dar se dovedește a nu fi adevărată în majoritatea calculelor mari. Această falsă intuiție a provenit dintr-o neînțelegere a modului în care ar fi utilizate procesoarele paralele. Dificultatea chestiunii rezidă în felul în care se împarte calculul între procesoare: la început ar părea că cel mai bun mod de a împărți un calcul între cîteva procesoare este de a da fiecăruia spre execuție o parte diferită a programului. Aceasta funcționează într-o oarecare măsură, dar (așa cum sugerează analogia cu ziariștii sus-menționați

nată) suferă de aceleași neajunsuri ca și atribuirea unei sarcini unei echipe de oameni: o mare parte din simultaneitatea potențială este pierdută în problemele asociate cu coordonarea. A programa un calculator prin fragmentarea programului este ca și cum o echipă mare de oameni ar vopsi un gard însărcinînd o persoană cu deschiderea cutiei de vopsea, o alta cu pregătirea suprafeței, alta cu aplicarea vopselei și alta cu curățarea pensulelor. Această fragmentare a funcțiilor necesită un grad înalt de coordonare și, după un anumit punct, adăugarea mai multor oameni nu ajută la accelerarea execuției sarcinii.

Un mod mai eficient de a folosi un calculator paralel este ca fiecare procesor să execute sarcini asemănătoare, dar pe secțiuni diferite de date. Această așa-numită *descompunere în paralel a datelor* sarcinii este precum vopsirea gardului prin atribuirea unei porțiuni separate a gardului fiecărui muncitor. Nu toate problemele se fragmentează atît de ușor ca vopsitul gardului, dar, în ceea ce privește calculele mari, această metodă funcționează surprinzător de bine. De exemplu, sarcinile de prelucrare a imaginii pot fi descompuse în părți simultane prin atribuirea fiecărui procesor a cîte unui petic mic din imagine. O problemă de căutare, precum jocul de șah, poate fi descompusă punînd procesoarele să caute concomitent fiecare o linie de joc diferită. În aceste exemple, gradul de accelerare realizat este aproape proporțional cu numărul de procesoare — așadar, cu cît mai multe, cu atît mai bine. Un timp suplimentar mic trebuie cheltuit cu împărțirea problemei între procesoare și cu colectarea răspunsurilor, dar, dacă problema este mare, calculul poate fi executat foarte eficient, chiar și pe o mașină paralelă cu zeci de mii de procesoare.

Calculele pe care tocmai le-am descris pot fi descompuse aproape evident pentru a rula în paralel, dar descompunerea în paralel a datelor funcționează de asemenea la sarcini mai complicate. Există surprinzător de puține probleme mari care nu pot fi tratate eficient prin prelucrarea paralelă. Chiar probleme pe care cei mai mulți oameni le consideră fatalmente secvențiale pot fi de obicei soluționate eficient pe un calculator paralel. Un exemplu este *problema urmăririi în lanț.* Copiii mei joacă un joc numit „căutarea comorii", care este bazat pe problema urmăririi în lanț. Le dau o bucată de hîrtie cu un indiciu despre locul unde este ascuns următorul indiciu. Acel indiciu conduce la următorul indiciu și așa mai departe, pînă ce ei ajung în sfîrșit la comoară. În versiunea pe calculator a acestui joc, programului i se dă ca intrare adresa unei locații de memorie conținînd adresa unei alte locații. Această locație conține adresa alteia și așa mai departe. În cele din urmă, adresa precizează o locație de memorie conținînd un cuvînt special care indică sfîrșitul lanțului. Problema este de a găsi ultima locație pornind de la prima.

Inițial, problema urmăririi în lanț pare ca un rezumat al unui calcul fatalmente secvențial, deoarece pare a nu exista nici o posibilitate pentru calculator de a găsi ultima locație din lanț fără a urmări adresele înlănțuite prin întreg șirul. Calculatorul trebuie să cerceteze prima locație pentru a găsi adresa celei de a doua, apoi cercetează a doua pentru a găsi adresa celei de a treia și așa mai departe. Totuși se dovedește că problema poate fi rezolvată în paralel. Un calculator paralel cu un milion de procesoare poate găsi ultimul element dintr-un lanț de un milion de adrese în douăzeci de pași.

Trucul este de a înjumătăți problema la fiecare pas, oarecum asemănător cu abordarea folosită în algoritmii de ordonare din capitolul cinci. Să presupunem că fiecare din cele un milion de locații de memorie are propriul său procesor, care poate trimite un mesaj oricărui alt procesor. Pentru a găsi sfîrșitul lanțului, fiecare procesor începe prin a trimite propria adresă procesorului care îi urmează în lanț — adică procesorului a cărui adresă este stocată în locația sa de memorie. Fiecare procesor cunoaște atunci adresa nu numai a procesorului care îi urmează, dar și a celui care îl precedă. Procesorul folosește această informație pentru a trimite adresa succesorului său către predecesorul său. Acum fiecare procesor cunoaște adresa procesorului situat cu două locuri înaintea sa în lanț, așa că acum lanțul unind primul procesor cu ultimul procesor are lungimea pe jumătate din cît fusese inițial. Acest pas de reducere este apoi repetat și de fiecare dată cînd este repetat, lungimea lanțului se înjumătățește. După douăzeci de iterații ale pasului de reducere, primul procesor dintr-un lanț de un milion de memorii cunoaște adresa ultimului. Metode asemănătoare pot fi aplicate multor alte sarcini care par inerent secvențiale.

Cînd scriu aceste rînduri, calculatoarele paralele sînt încă relativ noi și încă nu înțelegem ce alte tipuri de probleme pot fi descompuse pentru a profita eficient de multe procesoare. O regulă empirică pare a fi că paralelismul funcționează cel mai bine la probleme cu un mare volum de date, deoarece, dacă există multe elemente de date, există din abundență sarcini similare de împărțit între procesoare.

Un motiv pentru care cele mai multe calcule pot fi descompuse în subprobleme simultane este acela că cele mai multe calcule sînt modele ale lumii fizice. Cal-

culele bazate pe modele fizice pot opera în paralel deoarece lumea fizică funcționează în paralel. Imaginile grafice pe calculator, de exemplu, sînt deseori sintetizate printr-un algoritm care simulează procesul fizic al reflectării luminii de suprafețele obiectelor. Imaginea este dedusă dintr-o descriere matematică a formelor obiectelor calculînd cum ar ricoșa fiecare rază de lumină din suprafață în suprafață în timpul călătoriei de la sursă la ochi. Calcularea tuturor razelor de lumină poate avea loc concomitent, deoarece în lumea fizică reflectarea luminii are loc concomitent.

Un exemplu tipic al genului de calcul bine adecvat unui calculator paralel este o simulare a atmosferei, folosită în prognoza vremii. Matricea tridimensională de numere ce reprezintă atmosfera este asemănătoare cu spațiul fizic tridimensional. Fiecare număr precizează parametrii fizici ai unui anumit volum de aer — de exemplu, presiunea atmosferei într-un cub cu latura de un kilometru. Fiecare din aceste cuburi va fi reprezentat de cîteva numere precizînd valoarea medie pentru temperatură, presiune, viteza vîntului și umiditate. Pentru a prognoza cum va evolua volumul de aer dintr-un asemenea cub, ordinatorul calculează cum curge aerul între cuburi vecine: de exemplu, dacă într-un volum curge mai mult aer decît se scurge din el, atunci presiunea în acel volum va crește. Ordinatorul calculează de asemenea modificările susceptibile a fi atribuite unor factori precum lumina solară și evaporarea. Simularea atmosferei este calculată într-o serie de pași, fiecare din ei corespunzînd unei creșteri mici a timpului — să spunem o jumătate de oră — astfel încît fluxul simulat de aer și apă între celulele matricei este asemănător fluxului real de aer și apă în modelul atmosferic.

Rezultatul este un soi de film tridimensional — un film care se desfăşoară în concordanţă cu legile fizicii.

Desigur, precizia acestei simulări va depinde atît de rezoluţia, cît şi de precizia imaginii tridimensionale, ceea ce explică inexactitatea bine cunoscută a prognozelor meteo în timp. Dacă rezoluţia modelului este crescută şi condiţiile iniţiale sînt măsurate mai precis, atunci prognoza va fi mai bună — deşi nici o rezoluţie foarte înaltă nu va fi perfectă pe o perioadă lungă de timp, deoarece starea iniţială a atmosferei nu poate fi măsurată cu exactitate. Ca şi jocul la ruletă, sistemele meteo sînt haotice, astfel că o mică schimbare în condiţiile iniţiale va produce o modificare semnificativă în rezultat. Pe un calculator paralel, fiecărui procesor îi poate fi atribuită responsabilitatea prognozării vremii pe o zonă minusculă. Cînd bate vîntul de la o zonă la o alta învecinată, atunci procesoarele ce modelează aceste zone trebuie să comunice. Procesoarele modelînd zone separate geografic pot continua aproape independent, în paralel, deoarece vremea în aceste zone este aproape independentă. Calculul este local şi concomitent, deoarece fizica ce determină vremea este de asemenea locală şi funcţionează simultan peste tot.

Pe cînd simularea vremii este legată într-un mod evident de legi fizice, multe alte calcule sînt legate mai subtil de lumea fizică. De exemplu, calcularea notelor telefonice este simultană, deoarece telefoanele (şi abonaţii telefonici) operează independent în lumea fizică. Singurele probleme pe care nu ştim să le rezolvăm eficient pe un calculator paralel sînt cele pentru care dimensiunea în creştere a problemei este proporţională cu trecerea timpului. Un exemplu este problema predicţiei poziţiilor viitoare ale planetelor. (Ca o ironie a sorţii, aceasta este însăşi problema pentru care multe din

instrumentele noastre matematice de calcul au fost inventate inițial.)

Traiectoriile planetelor sînt consecința legilor bine definite ale impulsului și ale interacțiunilor gravitaționale între cele nouă planete și Soare. (Pentru simplitate, vom ignora efectele corpurilor mici, cum ar fi sateliții și asteroizii.) Toate informațiile necesare pentru a rezolva problema pot fi reprezentate cu nouă coordonate, deci nu există multe date. Dificultatea de calcul a problemei provine din faptul că, din cîte știm noi, calculul trebuie făcut prin determinarea pozițiilor succesive ale planetelor la fiecare din cele cîteva miliarde de pași minusculi, fiecare reprezentînd o scurtă perioadă de timp. Singurul mod pe care îl cunoaștem pentru a calcula pozițiile planetelor în viitor peste un milion de ani este de a calcula pozițiile lor la fiecare moment intermediar de acum și pînă atunci. Nu am cunoștință să existe un truc pentru a rezolva în paralel această problemă, truc asemeni celui folosit în problema urmăririi în lanț. Pe de altă parte, din cîte știu, nimeni nu a demonstrat că problema orbitei este intrinsec secvențială. Aceasta rămîne încă sub semnul întrebării.

Calculatoarele cu grad înalt de paralelism sînt acum destul de obișnuite. Ele sînt folosite cel mai adesea în calcule numerice foarte mari (precum simularea vremii) sau în calcule cu baze de date mari, cum ar fi extragerea datelor de vînzare din tranzacții cu cărți de credit. Deoarece calculatoarele paralele sînt construite din aceleași componente ca și calculatoarele personale, este probabil ca ele să devină cu timpul mai puțin costisitoare și mai răspîndite. Unul din cele mai interesante calculatoare paralele de astăzi este cel ce se ivește aproape accidental prin interconectarea mașinilor seriale. Rețeaua mondială de calculatoare numită internet este

încă folosită în principal ca un sistem de comunicare pentru oameni. Calculatoarele funcționează cel mai adesea ca un mediu — stocînd și furnizînd informații (ca poșta electronică) care au semnificație doar pentru oameni. Sînt convins că aceasta se va schimba. Deja au început să apară standarde ce permit acestor calculatoare să schimbe între ele atît programe, cît și date. Calculatoarele din internet, lucrînd împreună, au o capacitate de calcul potențială care depășește de departe orice calculator individual care a fost construit vreodată.

Cred că în cele din urmă internetul se va dezvolta pentru a include calculatoarele conținute în sisteme telefonice, automobile sau aparate electrocasnice simple. Astfel de mașini vor citi intrările lor direct din lumea fizică în loc de a se baza pe oameni ca intermediari. Pe măsură ce informația disponibilă pe internet devine tot mai bogată și tipurile de interacțiune dintre calculatoarele interconectate devin tot mai complexe, mă aștept că internetul va începe să ilustreze un *comportament nou*, dincolo de orice a fost programat explicit în sistem. De fapt, internetul începe deja să dea semne de comportament nou, dar pînă acum cea mai mare parte a sa este destul de simplă: epidemii de virusuri informatice și modele imprevizibile de comutare a mesajelor. Bănuiesc că, pe măsură ce calculatoarele din rețea încep să facă schimb de programe ce interacționează în loc de a schimba numai mesaje electronice, internetul va începe să se comporte mai puțin ca o rețea și mai mult asemeni unui calculator paralel. Bănuiesc că va fi cu mult mai interesant comportamentul nou al internetului.

Calculatoare care învață și se adaptează

Programele de calculator pe care le-am descris pînă acum funcționează în conformitate cu reguli fixe furnizate de programator. Ele nu au nici un mod de a inventa ele însele reguli noi sau de a le îmbunătăți pe cele care le sînt date. Programele de jucat șah vor continua să facă aceleași greșeli în repetate rînduri, indiferent cîte partide de șah joacă, dacă programatorii lor nu le depanează. În acest sens, calculatoarele sînt complet previzibile; în acest sens, calculatoarele „fac numai ceea ce sînt programate să facă" — un aspect deseori subliniat de apărătorii omenirii în disputa „om–mașină".

Dar nu tot software-ul este atît de inflexibil. Este posibil să se scrie programe care se perfecționează pe măsura acumulării experienței. Cînd funcționează cu astfel de programe, calculatoarele pot învăța din greșelile lor și pot corecta propriile erori. Acest lucru îl realizează folosind *conexiunea inversă* (*feedback*). Orice sistem bazat pe conexiune inversă are nevoie de trei feluri de informații:

1. Care este starea dorită (*scopul*)?
2. Care este diferența dintre starea actuală și starea dorită (*eroarea*)?

3. Ce acțiuni vor reduce diferența dintre starea actuală și starea țintă (*răspunsul*)?

Pentru a atinge scopul, sistemele cu conexiune inversă reglează răspunsul în conformitate cu eroarea. Cele mai simple și cele mai familiare exemple de sisteme cu conexiune inversă nu sînt sistemele de învățare, ci sistemele de control; termostatul din locuință este un bun exemplu. Acest sistem cu conexiune inversă recunoaște numai două erori posibile și produce numai două răspunsuri posibile. Scopul este menținerea unei anume temperaturi și erorile posibile sînt temperatura prea ridicată sau temperatura prea scăzută. Răspunsurile sînt predeterminate: dacă temperatura este prea scăzută, răspunsul este să aprindă sistemul de încălzire; dacă temperatura este prea ridicată, răspunsul este să închidă sistemul de încălzire. Deoarece termostatul poate doar opri sau porni sistemul de încălzire, răspunsul nu are nimic de-a face cu mărimea erorii. (Acesta este un fapt pe care am încercat în repetate rînduri să îl explic diverșilor locatari ai casei mele, care insistă să rotim termostatul complet pînă la nouăzeci de grade ori de cîte ori este prea rece, în speranța că atmosfera se va încălzi cumva mai repede; această tactică nu ajută. Termostatul poate doar aprinde sistemul de încălzire, nu îl poate iuți.)

Totuși, în principiu nu există nici un motiv pentru care un termostat de la instalația de încălzire a casei nu poate să răspundă proporțional cu eroarea. Un astfel de sistem ar avea nevoie de o metodă de a regla ieșirea de la sistemul de încălzire, în loc doar de a porni sau opri încălzirea. Aparatul ar fi fără îndoială mai complicat și mai scump, dar ar asigura menținerea mai precisă a temperaturii. Astfel de termostate *cu control pro-*

porțional sînt folosite astăzi în sisteme care controlează procese industriale delicate. Unele aparate electrocasnice — unele modele de mașini de spălat japoneze, spre exemplu — folosesc de asemenea controlul proporțional (sau o aproximație a sa), o trăsătură distinctivă căreia adesea i se face reclamă sub numele de *logică vagă* (*fuzzy logic*).

Un alt exemplu de sistem care folosește controlul proporțional este sistemul de pilot automat care dirijează un avion. În acest caz, scopul este de a menține avionul îndreptat către o direcție dată. Un detector de direcție, cum ar fi un compas, măsoară eroarea față de direcția în care zboară avionul. Pilotul automat răspunde făcînd o ajustare a poziției cîrmei avionului, ajustare proporțională cu mărimea și direcția erorii. Astfel o ușoară deviere în rută va duce la o schimbare ușoară în poziția palonierului, dar o deviere majoră, cum ar fi o deviere laterală produsă de o schimbare a vîntului, va duce la o schimbare mare în poziționarea palonierului. Dacă sistemul de pilot automat nu ar utiliza controlul proporțional, ci ar împinge palonierul complet la stînga sau complet la dreapta în maniera termostatului casnic, avionul ar oscila încoace și încolo într-un mod incomod și probabil periculos.

În toate aceste sisteme cu conexiune inversă, legătura dintre eroare și răspuns este fixată. Sensibilitatea răspunsului este predeterminată de proiectul sistemului de control. Dar este de asemenea posibil a proiecta un sistem cu conexiune inversă chiar mai flexibil, în care răspunsul sistemului *se adaptează* cu timpul. În acest caz, parametrii unui sistem cu conexiune inversă inițial sînt reglați de un al doilea sistem cu conexiune inversă. Dacă cel de-al doilea sistem cu conexiune in-

versă se adaptează şi se îmbunătăţeşte în timp, se poate spune că sistemul „a învăţat“ parametrii de control.

Să considerăm, de exemplu, cum învaţă să zboare cu un avion un pilot uman. Elevul-pilot tipic cîrmeşte exagerat la început — adică exagerează în corectarea fiecărei erori. Pilotul foloseşte ceva asemănător sistemului pe care îl foloseşte termostatul pentru a controla căldura: dacă avionul se abate prea mult spre stînga, răsuceşte spre dreapta; dacă se abate prea mult spre dreapta, răsuceşte spre stînga. Deoarece există o întîrziere între răsucirea manşei şi răspunsul avionului, sistemul începe să oscileze. Pilotul trebuie să înveţe cum să mişte palonierul proporţional cu eroarea, şi pentru aceasta este necesară aprecierea exactă a sensibilităţii răspunsului. Pilotul învaţă acest parametru printr-un alt sistem cu conexiune inversă; în acest caz, *scopul* conexiunii inverse este de a menţine avionul pe ruta corectă fără oscilaţii, iar *eroarea* este amplitudinea oscilaţiei. *Răspunsul* este reglarea răspunsului sistemului cu conexiune inversă primar — adică reglarea unghiului cu care este mişcată manşa pentru a corecta un anume unghi eronat în orientarea avionului. Ori de cîte ori primul sistem cu conexiune inversă al pilotului oscilează, pilotul reduce sensibilitatea sa. El îşi creşte sensibilitatea dacă avionul începe să devieze de la rută. De îndată ce pilotul învaţă sensibilitatea corectă, poate menţine avionul pe rută fără nici o oscilaţie.

Ar putea fi construit un pilot automat care foloseşte un al doilea sistem cu conexiune inversă pentru a regla propriii parametri, aşa cum s-a descris aici. În acest caz, s-ar putea spune că pilotul automat a „învăţat“ să zboare cu avionul, în acelaşi mod în care învaţă un pilot uman. Din cîte ştiu, astfel de sisteme de pilot automat care se adaptează nu sînt folosite în avioanele reale, dar,

dacă ar fi, ele ar avea anumite avantaje. Dacă avionul a suferit stricăciuni care au provocat o schimbare în manevrabilitatea avionului, cum ar fi un palonier parțial defect, pilotul automat ar fi în stare să se adapteze noii situații. Ar putea fi în stare chiar să se adapteze dacă legăturile la motorul ce controlează palonierul ar fi fost accidental inversate, astfel că un semnal care în mod normal rotește avionul spre dreapta îl rotește spre stînga. Ca și unui pilot uman, pilotului automat i-ar fi necesar un anume interval de timp satisfăcător pentru a se adapta unei schimbări atît de radicale de situație.

INSTRUIND CALCULATORUL

Noțiunea fundamentală de conexiune inversă este centrală în toate sistemele cu învățare, deși deseori ia o formă mai complicată decît pilotul automat cu autoreglare. Deseori, în programele de calculator conexiunea inversă este obținută prin *instruire* cu ajutorul exemplelor. Instructorul (de obicei o ființă umană) joacă rolul unui profesor, iar programul devine elev. Un exemplu clasic de sistem cu învățare este un program scris de Patrick Winston, un pionier al inteligenței artificiale, care învață definiția unor concepte precum „arc" dintr-o serie de exemple și contraexemple furnizate de instructor. Programul lui Winston învață concepte noi examinînd schițe sumare ale unor grămezi de blocuri. Programul este capabil să analizeze astfel de desene și să genereze descrieri simbolice ale grămezilor de blocuri: de exemplu, „două cuburi ce se ating, sprijinind o pană". Instructorul arată programului cîteva exemple de configurații de blocuri care

formează arce şi un alt set de exemple care nu formează, spunînd programului care sînt exemple de „arc" şi care nu sînt. Iniţial, programul nu are o definiţie pentru conceptul de „arc", dar, cînd i se arată aceste exemple şi contraexemple, începe să formuleze o definiţie provizorie. De fiecare dată cînd i se arată exemple noi, el testează definiţia sa provizorie pe noile exemple. Dacă definiţia descrie suficient de bine un exemplu sau elimină un contraexemplu, programul nu modifică definiţia. Dacă definiţia este eronată, ea este modificată pentru a se potrivi cu exemplul.

Aici avem de-a face cu un scenariu despre felul în care învaţă programul definiţia pentru „arc" din cîteva exemple. Să presupunem că primul exemplu ce i se arată programului este exemplul A din Figura 25, două blocuri dreptunghiulare verticale ce sprijină un triunghi. Pentru început, programul va trebui să facă o presupunere iniţială referitoare la definiţia arcului. Această estimare iniţială nu trebuie să fie exactă, deoarece ea va fi rafinată cu exemple viitoare. Să presupunem că programul foloseşte formele blocurilor pentru estimarea sa iniţială pentru o definiţie: „un arc constă din două blocuri dreptunghiulare şi un bloc triunghiular". Cel de-al doilea exemplu arătat programului ar putea fi aceleaşi blocuri, toate culcate (exemplul C din Figura 25). Acesta este un contraexemplu — adică un exemplu de ceva ce *nu* este un arc. Deoarece definiţia de lucru iniţială a programului a identificat eronat acest contraexemplu drept arc, el va modifica definiţia sa pentru a exclude exemplul. Programul face acest lucru prin identificarea diferenţelor dintre definiţie şi contraexemplu şi folosindu-le pentru a adăuga restricţii la definiţie. În acest caz, diferenţa este dată de relaţiile dintre blocuri, astfel că o definiţie îmbunătăţită va include aceste relaţii: „un

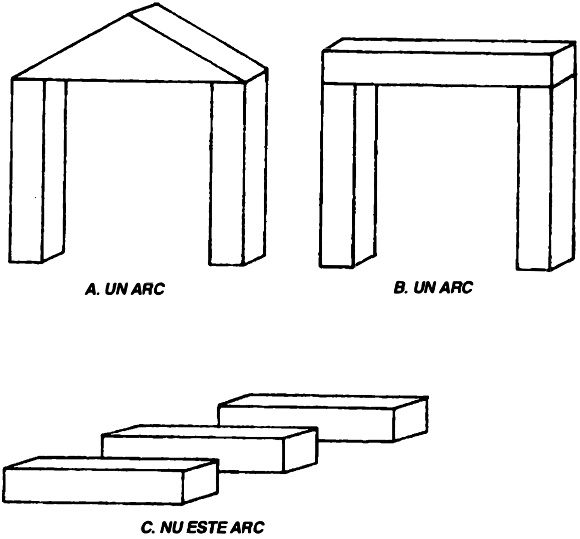


FIGURA 25 Exemple și contraexemple de arce

arc înseamnă două blocuri dreptunghiulare *verticale sprijinind* un bloc triunghiular". Acum să spunem că instructorul furnizeză un alt exemplu (B în Figura 25). Acest exemplu folosește în vîrf un bloc dreptunghiular în locul unuia triunghiular. Deoarece definiția de lucru a programului nu este suficient de cuprinzătoare pentru a include acest exemplu, programul va generaliza definiția sa pentru „arc" pentru a permite alte forme.

După ce i se arată aceste exemple și alte cîteva, programul va converge către următoarea definiție a unui arc: „un corp prismatic sprijinit de două blocuri verticale care nu se ating unul cu altul". Fiecare element al definiției a fost învățat făcînd un anumit fel de greșeală, iar definiția a fost modificată corespunzător. De îndată ce programul converge către definiția corectă, el înce-

tează a mai face greşeli şi lasă definiţia sa neschimbată. Apoi el poate identifica în mod corect drept arc orice arc care îi este arătat, chiar dacă nu a mai văzut vreodată anterior acea mulţime concretă de blocuri. El a învăţat conceptul de „arc".

REŢELE NEURONALE

Programul lui Winston învaţă conceptul de „arc", dar concepte ca „atingînd", „bloc triunghiular" şi „sprijin" au fost introduse în el de la început. Reprezentarea sa asupra lumii a fost proiectată anume pentru grămezi de blocuri. Căutarea unei scheme de reprezentare mai generale, universale, a condus mulţi cercetători la sisteme de calcul cu structuri analoage reţelelor conexe de neuroni biologici, aşa cum apar în creier. Un astfel de sistem este numit *reţea neuronală* artificială.

O reţea neuronală este o reţea simulată de neuroni artificiali. Această simulare poate fi efectuată pe orice fel de calculator, dar, deoarece neuronii artificiali pot funcţiona concomitent, un calculator paralel este locul cel mai natural pentru a o efectua. Fiecare neuron artificial are o ieşire şi un număr mare de intrări, poate chiar sute sau mii. În tipul cel mai obişnuit de reţea neuronală, semnalele între neuroni sînt binare — adică fie 0, fie 1. Ieşirea unui neuron poate fi legată la intrările multor altora. Fiecare intrare are un număr asociat ei, numit *ponderea* sa, care determină cît de mare este efectul intrării asupra singurei ieşiri a neuronului. Această pondere poate fi orice număr, pozitiv sau negativ. Ieşirea neuronului este astfel determinată de un vot al semnalelor venind la intrările sale, modificate de ponderile intrărilor. Neuronul calculează ieşirea sa prin

înmulțirea fiecărui semnal de intrare cu ponderea intrării și adunarea rezultatelor; cu alte cuvinte, el sumează ponderea tuturor intrărilor care primesc un semnal 1. Dacă suma ponderată atinge un anumit prag, ieșirea este 1, altfel ieșirea este 0.

Funcția unui neuron artificial corespunde, foarte aproximativ, funcției mai multor tipuri de neuroni reali din creier. Neuronii reali au de asemenea o ieșire și multe intrări, iar legăturile intrărilor, numite *sinapse*, au forțe diferite (corespunzînd ponderilor diferite ale intrărilor). Un semnal poate fie intensifica, fie inhiba excitația neuronului (corespunzînd ponderilor pozitive și celor negative) și neuronul va fi excitat cînd stimularea combinată a intrărilor sale este egală cu sau depășește un anumit prag. În acest sens un neuron artificial este asemănător unuia real. Există de asemenea multe privințe în care un neuron real este cu mult mai complicat decît unul artificial, dar acest neuron artificial simplu este suficient pentru construirea unui sistem capabil de învățare.

Primul lucru de observat în legătură cu neuronii artificiali este acela că ei pot fi folosiți pentru a executa operațiile **Și, Sau** și **Invers**. Un neuron implementează funcția **Sau** dacă pragul este 1 și fiecare intrare are ponderea mai mare sau egală cu 1. Un neuron cu un prag egal cu suma ponderilor va implementa funcția **Și**. Neuronii cu o singură intrare, cu pondere negativă și cu prag 0 vor implementa funcția **Invers**. Întrucît orice funcție logică poate fi construită combinînd funcțiile **Și, Sau** și **Invers**, o rețea de neuroni poate implementa orice funcție booleană. Neuronii artificiali sînt blocuri constructive universale.

Nu știm foarte mult despre modul în care funcționează creierul uman, dar se pare că în unele părți ale

creierului informaţiile noi sînt învăţate prin modificarea forţei sinapselor ce leagă neuronii. Cu siguranţă acesta este cazul organismelor inferioare pe care facem experienţe — de exemplu, melcii de mare. Melcii de mare pot fi învăţaţi să dea anumite răspunsuri condiţionate şi se poate arăta că ei învaţă răspunsul modificînd forţa legăturilor din sinapsele dintre neuroni. Presupunînd că învăţarea umană funcţionează în acelaşi fel, dumneavoastră modificaţi (sper) legăturile în creierul dumneavoastră la citirea acestei cărţi.

O reţea de neuroni artificiali poate „învăţa“ schimbînd ponderile legăturilor sale. Un bun exemplu este un tip foarte simplu de reţea neuronală numită *perceptron*, care poate învăţa să recunoască formele. Modul în care perceptronii învaţă arată cum funcţionează cele mai multe reţele neuronale. Un perceptron este o reţea cu două straturi de neuroni şi o singură ieşire. Fiecare intrare în primul strat este legată cu un senzor, cum ar fi un detector de lumină care măsoară strălucirea unui punct pe o imagine. Fiecare intrare în al doilea strat este legată cu o ieşire de pe primul strat, aşa cum se arată în Figura 26.

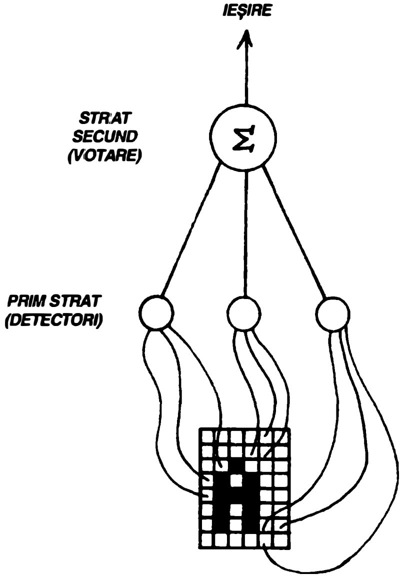
Să ne imaginăm că încercăm să învăţăm perceptronul să recunoască litera A, ceea ce vom realiza arătîndu-i un număr mare de exemple şi contraexemple pentru litera A. Pentru perceptron, scopul este de a regla ponderile celui de al doilea strat, astfel încît ieşirea sa va fi 1 dacă şi numai dacă îi este arătată imaginea unui A. Aceasta se realizează reglînd acele ponderi de cîte ori face o greşeală. Fiecare neuron din primul strat al perceptronului cercetează o zonă mică a indiferent cărui exemplu i se prezintă. Fiecare dintre aceşti neuroni din primul strat este programat să recunoască o trăsătură locală caracteristică, cum ar fi un colţ anume sau o linie

cu o anumită orientare; acest lucru îl face folosindu-se de ponderile fixate ale propriilor intrări. De exemplu, iată un model de ponderi de intrare pozitive şi negative pentru cîmpul senzorial al unui neuron din primul strat programat să recunoască un colţ, cum ar fi punctul din vîrful majusculei A:

-	-	-	-	-	-
-	-	+	-	-	-
-	+	+	+	-	-
-	+	+	+	+	-
+	+	+	+	+	+
+	+	+	+	+	+

Primul strat al perceptronului conţine mii de astfel de neuroni detectori de trăsături caracteristice,

FIGURA 26 Perceptron



fiecare programat să recunoască un anume tip de trăsătură locală într-o parte anume a cîmpului receptor. Acest prim strat de neuroni detectează în imagine trăsături care sînt folositoare pentru a distinge literele; piciorușele literelor sînt ușor de detectat, astfel încît ele fac literele mai ușor de recunoscut pentru perceptron, întocmai cum fac o anume literă mai ușor de identificat pentru ochiul omenesc.

Detectorii de trăsături caracteristice locale din primul strat aduc probele, iar ponderile stratului al doilea determină cum să cîntărim aceste probe. De exemplu, un colț îndreptat în sus în partea superioară a imaginii stă mărturie în favoarea unui A, în vreme ce un colț îndreptat în jos în partea centrală indică tocmai contrariul. Perceptronul învață prin reglarea ponderilor intrărilor din cel de-al doilea strat. Algoritmul de învățare este foarte simplu: de cîte ori instructorul indică perceptronului că a comis o greșeală, perceptronul va regla ponderile tuturor intrărilor care au votat în favoarea greșelii, astfel încît să facă viitoarele greșeli mai puțin probabile. De exemplu, dacă perceptronul identifică incorect o imagine ca fiind A, ponderile tuturor intrărilor care au votat în favoarea concluziei false vor fi micșorate. Dacă perceptronul nu izbutește să identifice un A veritabil, atunci intrările care au votat în favoarea lui A vor fi mărite. Dacă perceptronul are suficienți detectori de trăsături caracteristice de tipul corect, această metodă de pregătire va determina în cele din urmă perceptronul să învețe să recunoască literele A.

Procedeul de învățare a perceptronului este un alt exemplu de conexiune inversă. *Scopul* este de a fixa corect ponderile, *erorile* sînt identificări greșite ale exemplelor date pentru învățare, iar *răspunsul* este reglarea pon-

derilor. Observați că perceptronii, ca și programul lui Winston pentru arc, învață numai făcînd greșeli. Aceasta este o caracteristică a tuturor sistemelor cu învățare bazată pe conexiune inversă. Fiind asigurată o pregătire suficientă, această procedură concretă va converge întotdeauna către o alegere corectă a ponderilor, presupunînd că există o mulțime de ponderi care îndeplinește sarcina. Pare că perceptronul seamănă cu o mașină perfectă pentru recunoașterea formelor, dar capcana stă în presupunerea că există un model corect de ponderi care va îndeplini sarcina. Pentru a recunoaște litera A în poziții, mărimi și caractere de literă variate, perceptronul are nevoie de o mulțime foarte bogată de detectori de trăsături caracteristice în primul strat.

Perceptronii pot învăța să recunoască orice literă dacă le sînt date suficiente trăsături caracteristice cu care să lucreze, dar sînt unele tipuri de forme, mai complexe decît literele, care nu pot fi recunoscute în nici un fel prin însumarea unor trăsături caracteristice locale. De exemplu, un perceptron nu poate spune numai prin însumarea indiciilor din petice locale dacă toate punctele întunecate dintr-o imagine sînt sau nu conectate, deoarece conectivitatea este o proprietate globală; nici o însușire locală nu poate sta mărturie prin ea însăși pentru sau contra conectivității. Figura 27, adaptată după cartea *Perceptroni* a lui Marvin Minsky și Seymour Papert, demonstrează că nu întotdeauna conectivitatea poate fi stabilită doar cercetînd trăsături locale.

Din aceste motive, dar și din altele, perceptronii cu două straturi nu sînt cele mai practice rețele neuronale pentru recunoașterea celor mai multe tipuri de forme. Rețele neuronale mai generale, cu mai multe straturi, sînt capabile să recunoască forme mai complicate. Astfel

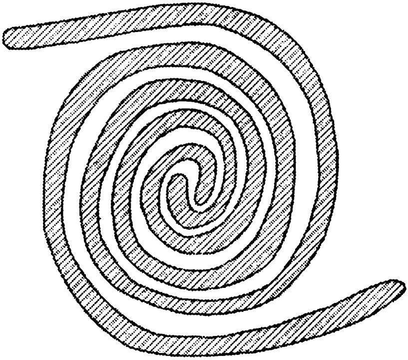


Figura 27 Perceptron în spirală

de rețele folosesc proceduri de învățare asemănătoare. Rețele neuronale cu învățare de acest tip sînt deseori folosite pentru sarcini ca recunoașterea imaginilor și recunoașterea vorbirii — sarcini care sînt dificil de descris precis printr-o mulțime fixată de reguli. De exemplu, sistemele simple pentru recunoașterea cuvintelor, care sînt montate astăzi în multe jucării pentru copii, se bazează pe rețele neuronale.

SISTEME CU AUTOREGLARE

Un sistem cu învățare pe bază de exemple și contraexemple are dezavantajul că necesită un instructor pentru a clasifica exemplele. Există un alt tip de rețea neuronală, care nu are nevoie de instructor — sau, exprimat altfel, există rețele în care semnalele pentru instruire sînt generate de rețeaua însăși. O astfel de rețea cu autoinstruire este un *sistem cu autoreglare*. Sistemele cu autoreglare au fost studiate de mulți ani (Alan Turing a publicat o lucrare importantă în acest domeniu),

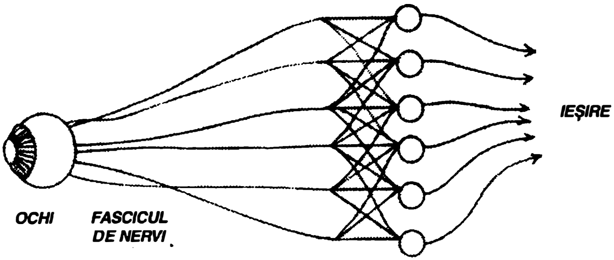


FIGURA 28 Ochi cu fascicol de nervi alterat și corector

dar recent a fost reluată activitatea de cercetare a unor astfel de sisteme și există chiar anumite progrese, datorate în parte folosirii calculatoarelor rapide. Ca și rețelele neuronale cu învățare, sistemele cu autoreglare se potrivesc în mod firesc calculatoarelor paralele.

Ca exemplu de sistem cu autoreglare care funcționează, să considerăm problema transmiterii unei imagini de la ochi la creier (*cf.* Figura 28). Retina, pe care este proiectată imaginea, este un strat bidimensional de neuroni sensibili la lumină. Imaginea de pe retină este transformată într-o imagine asemănătoare proiectată pe creier de un fascicul de neuroni care transferă imaginea. Dacă acest fascicul este conectat imperfect, atunci imaginea proiectată va fi ușor alterată, cu fiecare punct într-un loc ușor greșit. Voi descrie o rețea neuronală artificială cu autoreglare care poate învăța să corecteze o astfel de imagine, restabilind fiecare punct în poziția sa corectă. Corectorul constă dintr-un singur strat de neuroni aranjați într-o matrice bidimensională. Ieșirile acestor neuroni formează imaginea corectată. Dacă imaginea este doar puțin alterată, atunci fiecare punct

din imaginea alterată va fi în general în vecinătatea poziției sale corecte. Intrările fiecărui neuron explorează o vecinătate a punctelor din imaginea alterată și neuronul învață care din aceste puncte ar trebui legate cu ieșirea pentru a produce imaginea nealterată. Neuronul realizează legături fixînd ponderea intrării corecte la 1 și ponderea tuturor celorlalte intrări ale sale la 0.

Algoritmul de învățare pentru corector se bazează pe faptul că imaginile au o structură nealeatoare. Cum am discutat anterior, imaginile reale nu sînt doar tablouri aleatoare de puncte, ci imagini ale lumii, astfel că zonele apropiate ale imaginii tind să arate asemănător. Corectorul inversează această afirmație, presupunînd că punctele care tind să arate asemănător ar trebui prin urmare să fie aproape unul de altul. Neuronii din corector funcționează măsurînd corelația fiecăreia din intrările lor cu ieșirile neuronilor învecinați în timpul expunerii unei serii de imagini. De cîte ori un neuron face o greșeală excitînd diferit de vecinii săi, neuronul crește ponderea intrărilor care se potrivesc cu ieșirile vecinilor săi și descrește ponderile celorlalte intrări. Desigur, și vecinii săi învață legăturile lor în același timp, astfel că la început ne aflăm într-o situație în care (așa zicînd) orbul conduce orbul, dar în cele din urmă cîțiva dintre neuronii corectorului încep să blocheze intrările lor corecte și astfel ei devin instructori eficienți pentru vecinii lor. Din nou, singurii neuroni care sînt reglați sînt cei care au comis greșeli. În timp ce neuronii se instruiesc unul pe altul, o imagine nealterată începe să apară la ieșiri, iar în cele din urmă rețeaua se organizează ea însăși astfel încît să producă o imagine de claritate perfectă.

Pilotul automat cu autoreglare, programul lui Patrick Winston pentru arc, perceptronul și corectorul sînt

doar cîteva exemple de sisteme care învață. Toate aceste sisteme se bazează pe conexiunile lor inverse, interne și externe, și toate învață corectîndu-și greșelile. Proiectul fiecăruia din aceste sisteme a fost inspirat de un sistem biologic cu funcție asemănătoare. Culegînd roadele acestor produse ale evoluției, sîntem precum nebunul din fabula lui Esop *Gîsca cu ouă de aur*, care alege ouăle în loc de gîscă. În capitolul următor vom discuta despre gîscă.

CAPITOLUL 9

Dincolo de inginerie

Conform legendei, omul de știință și călugărul din secolul al treisprezecelea Roger Bacon a fost un diletant în magia neagră, și o dată a construit un cap mecanic vorbitor. Se spune că el dorea să apere Anglia de invadatori înălțînd un zid în jurul regatului și că a construit capul pentru a-i cere sfatul în legătură cu felul în care să ridice zidul. Bacon a fasonat capul din alamă, copiind modelul unui cap uman în cele mai mici amănunte. L-a încălzit la foc în timp ce rostea un descîntec — proces care a durat zile întregi. În cele din urmă capul a prins viață și a început să vorbească. Din nefericire, în acel moment Bacon era atît de epuizat de rostirea formulelor magice încît adormise. Tînărul său ucenic nu a dorit să-și trezească maestrul numai pentru niște discursuri incoerente ale unui cap de alamă, iar capul a explodat deasupra focului înainte ca Bacon să îi fi pus vreo întrebare.

Legenda lui Bacon are elemente în comun cu poveștile despre alți magicieni care au construit o inteligență artificială: Dedal, Pigmalion, Albert cel Mare, Rabinul din Praga. O temă comună multora din aceste povești este aceea că o formă de a găti sau a coace este necesară pentru a face ceva să înceapă a gîndi. În epoca dinain-

tea mașinilor de calcul, puțini și-au imaginat că un proces atît de complex precum gîndirea ar putea fi vreodată fragmentat în operații care ar putea fi implementate de mecanisme. În loc de acesta, se presupunea că, dacă inteligența ar fi fost vreodată creată, aceasta s-ar fi petrecut printr-un proces evolutiv — adică printr-un proces în care comportarea complexă apare ca o consecință globală a miliardelor de interacțiuni locale minuscule. S-a presupus că vrăjitorul nu ar avea nevoie de diagrama corectă de legături, ci de rețeta corectă, după care ingredientele s-ar fi organizat *ele însele* într-o inteligență. Un astfel de proces ar permite crearea unei inteligențe fără ca vrăjitorul să înțeleagă exact cum funcționează procesul sau inteligența însăși.

Destul de ciudat, în linii fundamentale sînt de acord cu această concepție preștiințifică: eu cred că sîntem capabili să creăm o inteligență artificială și presupun că procesul de creație va fi unul în care aranjăm ca inteligența să apară dintr-o serie complexă de interacțiuni pe care nu le înțelegem în amănunt — adică printr-un proces mai puțin asemănător construirii unei mașini și mai asemănător coptului unei prăjituri sau creșterii unei grădini. Noi nu vom construi inginerește inteligența artificială; mai degrabă vom aranja condițiile potrivite în care se poate ivi o inteligență. Cea mai mare realizare a tehnologiei noastre ar putea foarte bine fi crearea instrumentelor care să ne permită să trecem *dincolo* de inginerie — care să ne permită să creăm mai mult decît putem înțelege.

Înainte de a discuta cum ar putea funcționa acest proces emergent de proiectare, să considerăm cel mai bun exemplu de inteligență pe care îl avem: creierul uman. Deoarece creierul însuși a fost „proiectat“ — prin procesul emergent al evoluției darwiniste — poate fi folo-

sitoare comparaţia cu proiectele inginereşti pe care le-am considerat pînă acum.

CREIERUL

Creierul uman are aproximativ 10^{12} neuroni şi fiecare neuron are în medie 10^5 legături. Creierul este într-un anumit grad un sistem cu autoreglare, dar ar fi greşit să îl privim ca o masă omogenă. El conţine sute de tipuri diferite de neuroni, dintre care mulţi apar doar în anumite regiuni. Studii ale ţesutului cerebral arată că modelele legăturilor neuronale diferă, de asemenea, în regiuni diverse ale creierului: există cam cincizeci de zone în care modelul este recunoscut ca fiind diferit, şi probabil există mult mai multe în care diferenţele în anatomia neuronală sînt, pentru noi, prea subtile pentru a le distinge.

Fiecare zonă a creierului este aparent specializată pentru un tip specific de funcţie, cum ar fi recunoaşterea culorii în imagini vizuale, producerea intonaţiei în vorbire sau reţinerea numelor lucrurilor. Ştim asta pentru că, atunci cînd anumite zone sînt vătămate de un accident sau de o lovitură, există o pierdere de funcţie corespunzătoare. De exemplu, vătămarea zonelor 44 şi 45 din partea stîngă a lobului frontal — alcătuiesc împreună zona lui Broca — conduce deseori la pierderea capacităţii de a vorbi corect gramatical. Persoanele astfel afectate pot încă pronunţa clar cuvinte şi pot înţelege vorbirea altora, dar ele vor fi incapabile să construiască propoziţii corecte gramatical. Vătămarea unei zone cunoscute sub numele de convoluţia inelară, situată puţin mai departe către partea posterioară a creierului, provoacă dificultăţi la citit şi scris; vătămarea altor

zone are ca efect incapacitatea pacientului de a-și reaminti denumirile obiectelor familiare sau de a recunoaște fețe familiare.

Ar fi greșit să presupunem că zonele diferite ale creierului sînt analoage blocurilor constructive funcționale ale unui calculator. Pe de o parte, vătămarea celor mai multe din aceste zone nu va provoca o pierdere precisă de funcționalitate: înlăturarea celei mai mari părți din lobul frontal drept, de exemplu, uneori provoacă schimbări greu de definit în personalitate și uneori nu provoacă nici o schimbare perceptibilă. Chiar și în acele cazuri în care pierderea de funcționalitate este precisă, nu este deloc evident că funcția a fost efectuată în întregime în zona vătămată; s-ar putea ca zona doar să fi asigurat un element minor de sprijin necesar funcției. Un automobil cu o baterie descărcată nu va fi în stare să se miște, dar de aici nu deducem că bateria este răspunzătoare pentru propulsia mașinii.

Sînt anumite zone ale creierului — în special zone aflate lîngă partea din spate a capului, asociate cu prelucrarea vizuală — în care putem înțelege de fapt modelul legăturilor: de exemplu, cele implicate în receptarea intrărilor de la ochiul stîng și ochiul drept pentru a crea senzația de profunzime în vederea stereoscopică. Dar pentru cea mai mare parte a creierului „modelul legăturilor“ rămîne un mister. Chiar părerea că în cea mai mare parte a creierului este întinsă o rețea de conexiuni pentru funcții specifice s-ar putea dovedi a fi incorectă. Limbajul, de exemplu, pare a fi prelucrat îndeosebi în partea stîngă, în vreme ce orientarea spațială, cum ar fi capacitatea de a înțelege o hartă, pare a fi executată în primul rînd în partea dreaptă. Și totuși, sub microscop modelul unui țesut din emisfera cere-

brală dreaptă şi din emisfera cerebrală stîngă arată aproape la fel. Dacă există o diferenţă sistematică în modelele legăturilor în cele două emisfere ale creierului, pentru noi ea este prea subtilă pentru a o discerne.

Poate că funcţiile creierului sînt învăţate într-un soi de proces cu autoreglare care schimbă forţa diverselor legături sinaptice pentru a adapta o zonă la un anume tip de funcţie. Această afirmaţie este cu siguranţă adevărată într-un anumit grad. Ştim, de exemplu, că o maimuţă cu un deget lipsă va continua să folosească zona din creierul său care în mod normal prelucrează informaţii despre acel deget: neuronii neocupaţi sînt recrutaţi pentru a realiza prelucrarea pentru celelalte degete ale animalului. Fiinţele umane probabil rearanjează funcţiile creierului într-un mod asemănător în cursul însănătoşirii după un atac cerebral. Victima unui atac cerebral poate avea iniţial dificultăţi cu o funcţie anume, cum ar fi recunoaşterea feţelor, iar apoi să înveţe din nou funcţia, cu timpul. Întrucît neuronii vătămaţi nu se pot regenera ei înşişi, probabil că pacientul învaţă din nou funcţia recrutînd neuroni dintr-o altă parte a creierului.

Dacă funcţii precum recunoaşterea feţelor şi înţelegerea limbajului sînt învăţate în părţi diferite ale creierului, atunci trebuie de asemenea ca într-un sens aceste funcţii să fie încorporate de la început. Copiii nou-născuţi sînt deosebit de interesaţi de feţe în primele zile de viaţă şi învaţă să le recunoască cu mult timp înainte de a învăţa să facă distincţie între forme mult mai simple, cum ar fi literele. De asemenea, copiii mici par predispuşi să urmărească cu atenţie anumite modele de vorbire care le permit să înveţe cuvinte şi gramatică. Funcţiile care prelucrează limbajul şi recunosc feţe vor ajunge în părţi diferite ale creierului, proba-

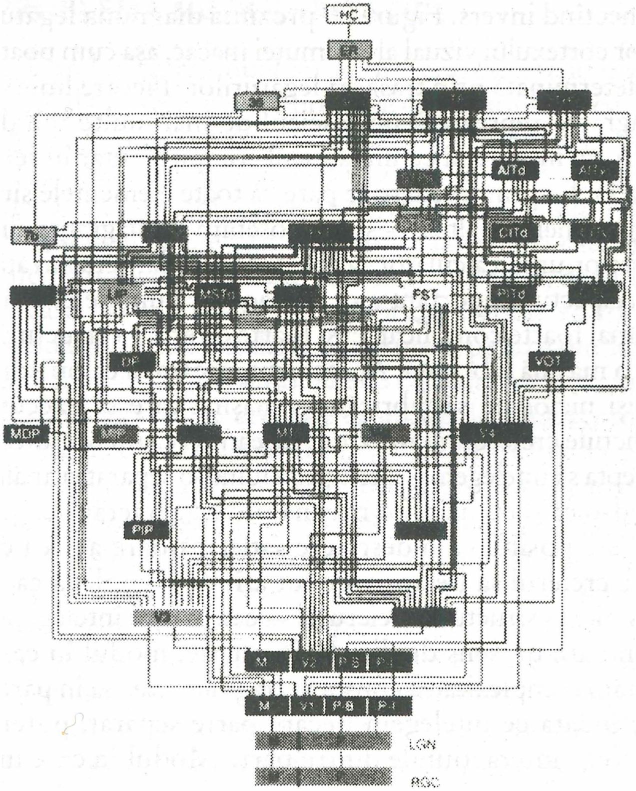


Figura 29 Diagrama bloc a cortexului vizual la macac

bil deoarece acele părți ale creierului sînt cumva pregătite să realizeze acele funcții diferite.

Chiar și în acele porțiuni ale creierului în care funcțiile par a fi implementate, modelul legăturilor prezintă puțină asemănare cu structura ierarhică a blocurilor funcționale dintr-un calculator: nu există un model simplu de intrări ducînd la ieșiri. În schimb, legăturile sînt deseori bidirecționale, cu o mulțime de neuroni conectînd într-un sens și o mulțime complementară

conectînd invers. Figura 29 prezintă diagrama legăturilor cortexului vizual al maimuței macac, așa cum poate fi determinată prin trasarea legăturilor. Fiecare linie în diagramă reprezintă un fascicul de mai multe mii de neuroni împreună cu un fascicul complementar în sens invers. La prima vedere se pare că toate elementele sînt legate unele cu altele — spre deosebire de diagrama circuitelor unui calculator fabricat, care e clar ierarhizată.

Aspectul important aici este acela că un creier e nu numai foarte complicat, ci și foarte diferit, ca structură, de o mașină fabricată. Aceasta nu înseamnă că nu vom reuși niciodată să fabricăm o mașină care să execute funcțiile creierului uman, dar înseamnă că nu ne putem aștepta să înțelegem o inteligență luînd-o separat și analizînd-o ca și cum ar fi o mașină proiectată ierarhic.

Este posibil ca o descriere satisfăcătoare a ceea ce face creierul să fie aproape tot atît de complexă ca și descrierea structurii creierului — caz în care înțelegerea sa nu are un sens deplin. În inginerie, modul în care tratăm complexitatea constă în fragmentarea sa în părți. De îndată ce înțelegem fiecare parte separat, putem înțelege interacțiunile dintre părți. Modul în care înțelegem fiecare parte constă în a aplica procesul ingineresc recursiv, rupînd fiecare parte în subpărți, și așa mai departe. Proiectul unui calculator electronic, împreună cu toate programele sale, este o mărturie impresionantă privind cît de departe poate fi împins acest proces. Cîtă vreme funcția fiecărei părți este precizată și implementată cu grijă, și atîta vreme cît interacțiunile dintre părți sînt controlate și previzibile, acest sistem de a „diviza și stăpîni“ funcționează foarte bine, dar un obiect evoluat precum creierul nu are în mod necesar acest tip de structură ierarhică.

PROBLEMA MODULARITĂȚII

Întrebuințarea unei structuri ierarhice stricte este călcîiul lui Ahile pentru procesul de fabricație, deoarece ea creează cu necesitate un soi de inflexibilitate de nezdruncinat pe care o asociem cu mașinile. Așa cum am discutat în capitolul șase, sistemele ierarhice sînt fragile în sensul că ele sînt predispuse la eșecuri catastrofale. Produsele fabricate sînt inerent fragile, deoarece fiecare parte a unui sistem fabricat trebuie să corespundă descrierii detaliate din proiect, pentru modul în care trebuie să interacționeze cu celelalte părți. Această descriere detaliată servește ca un soi de contract între componente. Dacă una dintre componente încalcă partea sa din contract, stipulațiile din proiectul sistemului nu sînt valabile, iar sistemul eșuează într-un mod imprevizibil. Pana unei singure componente de nivel inferior se poate propaga în sistem cu efecte catastrofale. Desigur, sistemele complexe, precum calculatoarele și avioanele, sînt fabricate pentru a evita aceste așa-numite avarii într-un singur loc prin metodele de redundanță descrise în capitolul șase, dar astfel de tehnici protejează sistemul numai de avarii prevăzute. Toate consecințele posibile ale unei avarii concrete trebuie prevăzute și înțelese — o sarcină care devine din ce în ce mai complexă.

Problema depășește avaria componentelor individuale. Într-un sistem complicat, chiar și părți funcționînd corect pot produce comportamente neașteptate atunci cînd interacționează. Deseori, cînd un sistem mare de programe funcționează necorespunzător, programatorii răspunzători pentru fiecare din părți pot demonstra convingător că fiecare din subrutinele lor face ceea ce trebuie. Deseori toți au dreptate, în sen-

sul că fiecare subrutină implementează corect funcția sa proprie precizată. Cusurul stă în descrierile detaliate a ceea ce se presupune că fac părțile și a felului în care se presupune că interacționează ele. Astfel de descrieri detaliate sînt dificil de scris corect fără a anticipa toate interacțiunile posibile. Sistemele complexe mari, precum sistemele de operare ale calculatoarelor sau rețelele telefonice, prezintă deseori comportamente deconcertante și neprevăzute, chiar și atunci cînd fiecare parte funcționează așa cum a fost proiectată. Poate vă amintiți că acum cîțiva ani liniile telefonice interurbane din estul Statelor Unite au încetat să mai transmită apeluri timp de cîteva ore. Sistemul folosea un proiect sofisticat, tolerant la greșeli, bazat pe redundanță. Toate componentele sale funcționau corect, dar o interacțiune neprevăzută între două versiuni ale programului rulînd la centrale diferite a provocat avaria întregului sistem.

Mi se pare uimitor că procesele inginerești funcționează atît de bine. Proiectarea a ceva atît de complicat precum un calculator sau un sistem de operare poate necesita mii de oameni. Dacă sistemul este suficient de complicat, nici o persoană nu poate avea imaginea completă a sistemului. Această situație duce în general la greșeli ce provin din înțelegerea greșită a relațiilor reciproce și din ineficiențele proiectului. Din nou, astfel de dificultăți de separare se înrăutățesc pe măsură ce sistemul devine mai complex.

Este important să observăm că problemele schițate anterior nu sînt slăbiciuni inerente ale mașinilor sau ale programelor *per se*. Ele sînt slăbiciuni ale procesului ingineresc de proiectare. Știm că nu tot ceea ce este complex este fragil. Creierul este mult mai complicat decît un calculator, și totuși este mai puțin predispus la eșecuri

catastrofale. Diferența de încredere dintre creier și calculator ilustrează deosebirea dintre produsele evoluției și cele ale ingineriei. O singură eroare într-un program de calculator poate provoca blocarea, dar creierul este de obicei capabil să tolereze idei greșite și informații incorecte și chiar componente cu funcționare necorespunzătoare. Neuronii individuali din creier mor mereu și nu sînt niciodată înlocuiți; dacă vătămarea nu este gravă, creierul reușește să se adapteze și să compenseze aceste lipsuri. (Ca o ironie a sorții, pe cînd scriam acest capitol, calculatorul meu s-a blocat și a trebuit repornit.) Oamenii rareori se blochează.

EVOLUȚIE SIMULATĂ

Așadar, în crearea unei inteligențe artificiale, care este alternativa la inginerie? O abordare posibilă este imitarea în calculator a procesului evoluției biologice. Evoluția simulată ne dă un mod diferit de a proiecta hardware și software complicate — un mod care evită multe din problemele ingineriei. Pentru a înțelege cum funcționeză evoluția simulată, să considerăm un exemplu concret. Să spunem că dorim să proiectăm o porțiune dintr-un program care ordonează descrescător numere. Abordarea inginerească standard ar fi să scriem un astfel de program folosind unul din algoritmii de ordonare discutați în capitolul cinci, dar în locul ei să considerăm cum am putea face să se „dezvolte" programul.

Primul pas este generarea unei „populații" aleatoare de programe. Putem crea această populație folosind un generator de numere pseudoaleatoare pentru a produce secvențe aleatoare de instrucțiuni (*cf.* capitolul

patru). Pentru a accelera procesul, putem folosi numai acele instrucțiuni utile pentru ordonare, cum ar fi instrucțiunile de comparare și de schimbare. Fiecare din aceste secvențe aleatoare de instrucțiuni este un program: populația aleatoare va conține, să spunem, zece mii de astfel de programe, fiecare lung de cîteva sute de instrucțiuni.

Următorul pas este testarea populației pentru a găsi care programe sînt cele mai reușite. Pentru aceasta este necesar să rulăm fiecare program ca să vedem dacă poate sau nu ordona corect un șir test. Desigur, întrucît programele sînt aleatoare, este posibil ca nici unul să nu treacă testul — dar din noroc chior unele vor ajunge mai aproape de o ordonare corectă decît altele. De exemplu, un program poate muta din întîmplare numerele mici la sfîrșitul șirului. Prin testarea fiecărui program pe cîteva șiruri diferite de numere, putem atribui fiecărui program o notă.

Următorul pas este crearea de noi populații descendente din programele cu notele cele mai mari. Pentru a realiza aceasta, programele cu notele sub medie sînt îndepărtate: numai programele cele mai adaptate supraviețuiesc. Noua populație este creată făcînd cópii ale programelor supraviețuitoare cu mici variații aleatoare, un proces analog reproducerii asexuate cu mutații. Pe de altă parte, putem „da naștere“ la programe noi împerechind supraviețuitori din generația precedentă — un proces analog cu reproducerea sexuată. Realizăm acest lucru combinînd secvențe de instrucțiuni din fiecare program „părinte“ pentru a produce un „copil“. Probabil că părinții au supraviețuit deoarece conțineau secvențe utile de instrucțiuni și există o bună șansă ca descendentul să moștenească de la fiecare părinte cele mai folositoare trăsături.

Cînd este produsă noua generație de programe, ea este din nou supusă aceleiași proceduri de testare și selecționare, astfel încît încă o dată programele cele mai adaptate supraviețuiesc și se reproduc. Un calculator paralel va produce o nouă generație la fiecare cîteva secunde, astfel că procesele de selecție și variație pot fi repetate convenabil de multe mii de ori. Cu fiecare generație, priceperea medie a populației tinde să crească — adică programele devin din ce în ce mai bune la ordonat. După cîteva mii de generații, programele vor ordona perfect.

Am folosit evoluția simulată pentru a dezvolta un program pentru rezolvarea unor probleme concrete de ordonare, așa încît știu că procesul funcționează așa cum l-am descris. În experiențele mele am favorizat de asemenea programele care ordonau repede șirurile test, astfel că era mai mare probabilitatea ca programele mai rapide să supraviețuiască. Acest proces evolutiv a creat programe de ordonare foarte rapide. Pentru problemele care mă interesau, programele care s-au dezvoltat erau de fapt puțin mai rapide decît oricare din algoritmii descriși în capitolul cinci — și de fapt ele erau mai rapide la ordonat numere decît oricare program pe care l-aș fi putut scrie eu însumi.

Unul din lucrurile interesante despre programele de ordonare care au apărut în experimentul meu este că nu înțeleg cum funcționează. Am examinat cu grijă secvențele lor de instrucțiuni, dar nu le înțeleg: nu am nici o explicație privind funcționarea programelor care să fie mai simplă decît înseși secvențele de instrucțiuni. Poate că programele nu pot fi înțelese — poate că nu există nici o modalitate de a fragmenta funcționarea programului într-o ierarhie de părți care pot fi înțelese. Dacă acesta este adevărul — evoluția poate

produce ceva atît de simplu ca un program de ordonare care este eminamente de neînțeles — aceasta nu prevestește nimic bun pentru perspectivele noastre de a înțelege vreodată creierul uman.

Am folosit teste matematice pentru a demonstra că programele de ordonare dezvoltate sînt fără greșeală, dar am chiar mai multă încredere în procesul care le-a produs decît în testele matematice. Aceasta deoarece știu că fiecare din programele de ordonare apărute descinde dintr-un șir lung de programe a căror supraviețuire a depins de capacitatea lor de a ordona.

Faptul că programul dezvoltat nu poate fi întotdeauna înțeles provoacă neîncrederea unor persoane în a-l folosi în aplicații reale, dar cred că această nervozitate este fondată pe presupuneri false. Una dintre aceste presupuneri este că sistemele inginerești sînt totdeauna bine înțelese, dar lucrul este adevărat numai pentru sisteme relativ simple. Așa cum am notat, nici o singură persoană nu înțelege complet un sistem de operare complex. A doua presupunere falsă este aceea că sistemele sînt mai puțin demne de încredere dacă nu pot fi explicate. Fiind posibilă alegerea între a zbura cu un avion pus în funcțiune de un program de calculator sau cu unul pilotat de un pilot uman, aș alege pilotul uman. Și aș face aceasta chiar dacă nu înțeleg cum lucrează pilotul uman. Prefer să am încredere în procesul care a produs pilotul. Știu că un pilot, ca și programele de ordonare, este descendent dintr-o linie lungă de supraviețuitori. Dacă siguranța avionului ar depinde de ordonarea corectă a numerelor, aș prefera să depind de un program de ordonare dezvoltat decît de unul scris de o echipă de programatori.

DEZVOLTÎND O MAȘINĂ CARE GÎNDEȘTE

Evoluția simulată nu este prin ea însăși o soluție la problema construirii unei mașini care să gîndescă, dar ne indică direcția corectă. Ideea fundamentală este de a deplasa povara complexității de pe ierarhia proiectului pe puterea combinatorie a calculatorului. În definitiv, evoluția simulată este un soi de tehnică de căutare euristică și care explorează spațiul proiectelor posibile. Regulile euristice pe care le folosește pentru a explora spațiul sînt: *încearcă un proiect asemănător cu cele mai bune proiecte pe care le-ai găsit pînă acum* și *combină elemente a două proiecte reușite*. Ambele euristici funcționează bine.

Evoluția simulată este o metodă bună pentru a crea structuri noi, dar este un mod ineficient pentru a adapta un proiect existent. Slăbiciunile sale, ca și punctele sale tari, provin din orbirea inerentă a evoluției pentru acel „de ce“ al unui proiect. Spre deosebire de sistemele cu conexiune inversă descrise în ultimul capitol, în care modificări specifice erau făcute pentru a corecta lipsuri concrete, evoluția alege variațiile orbește, fără a lua în considerare cum vor influența modificările rezultatul.

Creierul uman profită de ambele mecanisme: este în aceeași măsură un rezultat al învățării, cît și al evoluției. Evoluția trasează trăsăturile generale, iar dezvoltarea individualului în interacțiune cu mediul său înconjurător completează imaginea. De fapt, produsul evoluției nu este atît un proiect pentru creier, cît proiectul pentru un proces care generează un creier — nu atît un plan, cît o rețetă. Astfel, există niveluri multiple de procese emergente acționînd simultan. Un proces evolutiv creează o rețetă pentru dezvoltarea unui creier,

iar procesul de dezvoltare interacționează cu mediul înconjurător pentru a realiza conexiunile în creier. Procesul de dezvoltare include atît procese de morfogeneză determinate intern, cît și procesele externe de învățare. Forțele de maturizare ale morfogenezei determină celulele nervoase să crească în modele corecte și procesul de învățare reglează fin legăturile. Stadiul final în învățarea creierului este un proces cultural, în care cunoștințele dobîndite de alți indivizi de-a lungul multor generații sînt transferate în el.

Am descris fiecare dintre aceste mecanisme emergente (evoluție, morfogeneză, învățare) ca și cum ele ar fi procese discrete, dar în realitate ele se împletesc sinergetic. Nu există o linie clară de demarcație între forțele de maturizare ale morfogenezei și procesele instructive ale culturii. Cînd o mamă gîngurește copilului său nou-născut, acesta este atît un proces instructiv, cît și un sprijin în maturizarea creierului copilului. Procesul morfogenezei este el însuși un proces de adaptare, în care fiecare celulă se dezvoltă în interacțiune constantă cu restul celulelor organismului, într-un proces cu conexiune inversă complex, care tinde să corecteze erorile și să mențină direcția de dezvoltare a organismului.

Există de asemenea interacțiuni sinergetice între procesele evolutive care creează specia și procesele de dezvoltare care creează individul. Cel mai clar exemplu de interacțiune dintre dezvoltare și evoluție este cunoscut ca efectul Baldwin, descris pentru prima oară de biologul evoluționist James Baldwin în 1896 și redescoperit de informaticianul Geoffrey Hinton aproape un secol mai tîrziu. Ideea fundamentală a efectului Baldwin este că la combinarea evoluției cu dezvoltarea, evoluția poate avea loc mai repede; procesele de adap-

tare ale dezvoltării pot corecta defectele într-un proces evolutiv imperfect.

Pentru a înțelege efectul Baldwin, trebuie mai întîi apreciată dificultatea dezvoltării trăsăturilor pentru care este necesar ca mutații multiple să apară împreună. Să considerăm evoluția instinctului de a construi un cuib din comportamentul pasărilor. Este rezonabil să presupunem că pentru a construi un cuib este nevoie de cîteva zeci de pași individuali, cum ar fi găsirea unei crenguțe, apucarea ei cu ciocul, transportarea crenguței la cuib și așa mai departe. Să presupunem, de asemenea, de dragul exemplului, că fiecare din acești pași necesită o mutație diferită și că, pentru ca pasărea să obțină un avantaj (sub forma unui cuib terminat), este necesar întregul set de mutații. Cu alte cuvinte, dacă lipsește chiar și un singur pas, cuibul nu va fi construit și prin urmare pasărea nu va fi mai adaptată decît semenii săi și nu va obține nici un avantaj evolutiv. Evident, problema legată de dezvoltarea unei astfel de trăsături este că evoluția va selecta mutații pentru una din componentele sale numai dacă toate celelalte sînt prezente: apariția simultană a tuturor acestor mutații la un singur individ este un eveniment foarte puțin probabil. Deoarece nici un pas izolat nu este benefic doar prin el însuși, este dificil să ne imaginăm cum se poate dezvolta un comportament ca acela de a construi un cuib.

Efectul Baldwin este interacțiunea sinergetică dintre evoluție și învățare. Această interacțiune ajută la soluționarea acestei probleme recunoscînd parțial avantajul de care beneficiază pasărea printr-o mutație care produce un singur pas al sarcinii. O pasăre care se naște știind cum să facă unii pași va avea un avantaj asupra păsării care nu știe, deoarece va avea de învățat mai pu-

ţini paşi, astfel că este mai probabil să ajungă la un comportament care îi permite să construiască cu succes un cuib. Fiecare pas izolat cu care pasărea s-a născut contribuie la posibilitatea de a învăţa şi de aceea are o valoare în sine. Văzută în acest mod, fiecare mutaţie individuală va fi favorizată în mod independent, încît comportamentul legat de construcţia unui cuib va rezulta din paşi care sînt adăugaţi treptat la repertoriul de instincte al păsării în mai puţin timp decît ar fi necesar pentru un noroc probabilistic care produce deodată toate mutaţiile într-un singur individ. Efectiv, faptul că pasărea poate învăţa face ca evoluţia să se petreacă mai repede. Efectul Baldwin se aplică nu numai la învăţare, ci la orice mecanism de adaptare în dezvoltarea individului.

O parte din motivul pentru care sînt optimist în legătură cu perspectivele de a dezvolta o maşină care gîndeşte este că nu trebuie să pornim de la zero. Putem „umple" populaţia iniţială de maşini cu modele de structură pe care le observăm în creier. Putem, de asemenea, porni cu orice fel de modele de învăţare şi dezvoltare pe care le observăm în sistemele naturale, chiar dacă nu avem o înţelegere completă a lor. Aceasta ar trebui să ne ajute chiar dacă presupunerile noastre nu sînt întru totul exacte, deoarece a începe căutarea noastră de undeva din apropierea soluţiei este probabil mult mai bine decît să pornim la întîmplare. Incluzînd *ceva* dintr-un model de dezvoltare în acest proces, evoluţia unei maşini care gîndeşte ar putea profita de legea lui Baldwin.

Instruirea este un alt efect care reduce radical timpul necesar pentru a dezvolta un comportament complex. Un copil îşi dezvoltă inteligenţa cel puţin în parte pentru că are mulţi oameni de la care să înveţe. O parte

a acestei învățări este dobîndită prin pură imitare și o parte prin instruire explicită. Limbajul uman este un mecanism spectaculos pentru transferarea ideilor de la o minte la alta, permițîndu-ne acumularea cunoștințelor folositoare și a comportamentului de-a lungul multor generații cu o viteză care depășește de departe evoluția biologică. „Rețeta“ pentru inteligența umană stă la fel de mult în cultura umană ca și în genomul uman.

Totuși, chiar pornind cu tot ceea ce cunoaștem, nu m-aș aștepta ca noi să fim capabili să dezvoltăm inteligența artificială de nivel înalt într-un singur pas. Aici este o schiță sumară privind felul cum ar putea progresa șirul de stadii. Am începe prin a dezvolta un proiect al unei mașini cu inteligența unei insecte, să spunem, creînd un mediu simplu în care inteligența asemănătoare celei a insectei ar fi favorizată, și prin a porni cu o populație inițială predispusă, prin mecanismele sale de dezvoltare, să dezvolte tipurile de structuri neurale pe care le vedem la insecte. Printr-o succesiune de medii simulate, din ce în ce mai bogate, am putea în cele din urmă dezvolta inteligența noastră de la nivelul insectei la cel al unei broaște, al unui șoarece și așa mai departe. Chiar dacă pentru a merge atît de departe ar fi nevoie, fără îndoială, de zeci de ani de muncă și ar implica multe fundături și false începuturi, în cele din urmă această direcție de cercetare ar putea conduce la dezvoltarea unei inteligențe artificiale cu flexibilitatea și complexitatea creierului primatelor.

Dacă vom reuși vreodată să dezvoltăm o mașină care poate înțelege limbajul, am fi capabili că avansăm rapid, profitînd de cultura umană. Îmi imaginez că ar trebui să învățăm o mașină inteligentă printr-un proces asemănător celui prin care am învăța un copil, cu

acelaşi amestec de deprinderi, cunoştinţe, moravuri şi poveşti. Deoarece am încorpora cultură umană în reţeta maşinii pentru inteligenţă, maşina ce va rezulta nu va fi în întregime o inteligenţă artificială, ci mai degrabă o inteligenţă umană sprijinită de o minte artificială. Din acest motiv mă aştept să ne înţelegem bine cu ea.

Sînt conştient, desigur, că prin construirea unei astfel de maşini vom crea un nod de controverse morale. De exemplu, o dată ce o astfel de maşină va fi fost creată, va fi imoral să o stingem? Aş presupune că ar fi greşit să o stingem, dar nu pretind că sînt sigur de statutul moral al unui asemenea produs artificial inteligent. Din fericire, vom avea ani mulţi în care să rezolvăm astfel de chestiuni.

Cei mai mulţi oameni nu sînt interesaţi atît de chestiunile morale practice ale unui viitor ipotetic, cît de controversele filozofice pe care ni le pune nouă înşine simpla posibilitate a existenţei unei inteligenţe artificiale. Cei mai mulţi dintre noi nu se bucură să fie asemănaţi cu maşini. Este de înţeles: ar trebui să fim insultaţi de asemănarea cu maşini stupide, precum automobilele sau maşinile de prăjit pîine sau chiar calculatoarele de astăzi. A spune că mintea este ruda unui calculator din generaţia actuală este tot atît de înjositor precum a spune că o fiinţă umană este înrudită cu un melc. Şi totuşi, ambele afirmaţii sînt adevărate şi ambele pot fi de folos. Întocmai cum putem învăţa ceva despre noi înşine studiind structura neuronală a melcului, putem învăţa ceva despre noi înşine studiind caricatura simplă a gîndirii din calculatoarele de astăzi. Putem fi animale, dar într-un sens creierul nostru este un fel de maşină.

Mulţi dintre prietenii mei credincioşi sînt şocaţi că eu înţeleg creierul uman ca o maşină şi mintea ca un cal-

cul. Pe de altă parte, prietenii mei din lumea științifică mă acuză de misticism deoarece eu cred că nu vom putea niciodată atinge o înțelegere completă a fenomenului gîndirii. Și totuși, rămîn convins că nici religia, nici știința nu au înțeles totul. Bănuiesc că starea conștientă este o consecință a acțiunii legilor fizice normale și o manifestare a unui calcul complex, dar aceasta nu face pentru mine starea conștientă mai puțin misterioasă și minunată — mai degrabă o face cu atît mai mult astfel. Între semnalele neuronilor noștri și senzațiile gîndurilor noastre se întinde un gol atît de mare încît nu va putea fi niciodată acoperit de înțelegerea omenească. Astfel că atunci cînd spun despre creier că este o mașină, aceasta nu este o insultă pentru minte, ci recunoașterea potențialului unei mașini. Nu cred că o minte omenească este mai puțin decît ne-o imaginăm a fi, ci mai degrabă o mașină poate fi mult, mult mai mult.

LECTURI SUPLIMENTARE

HILLIS, W. DANIEL, *The Connection Machine*, The MIT Press Series in Artificial Intelligence, MIT, 1989.

KNUTH, DONALD ERVIN, *The Art of Computer Programming*, 4 volume, Addison-Wesley, 1997 și traduceri vol. I, Ed. Teora, 1999 și vol. II, Ed. Teora, 2000.

MINSKY, MARVIN LEE, *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.

PATTERSON, DAVID A. și JOHN L. HENNESSY, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufman Publishers, 1996.

WEINER, NORBERT, *Human Use of Human Beings: Cybernetics and Society*, Avon, 1986.

WINSTON, PATRICK HENRY, *Artificial Intelligence*, Addison-Wesley, 1998.

CUPRINS

Prefață: Magie în piatră 5
Mulțumiri 11
1 Piulițe și șuruburi 13
2 Blocuri constructive universale 35
3 Programare 57
4 Cît de universale sînt mașinile Turing? 81
5 Algoritmi și euristică 100
6 Memorie: informații și coduri secrete 117
7 Viteză: calculatoare paralele 136
8 Calculatoare care învață și se adaptează 153
9 Dincolo de inginerie 170
Lecturi suplimentare .190

Redactor
VLAD ZOGRAFI

Apărut 2001
BUCUREȘTI–ROMÂNIA

MULTIPRINT

Tipografia **MULTIPRINT** Iași
Calea Chișinăului 22, et 6, Iași 6600
tel. 032-211225, 236388 fax. 032-211252

SCIENCE **M**ASTERS

Seria SCIENCE MASTERS, publicată simultan în peste 25 de limbi, va cuprinde 12 cărți de difuzare a informației științifice într-o formă atractivă, accesibilă publicului larg. Oameni de știință eminenți din domenii ce merg de la astrofizică pînă la zoologie își fac cunoscute aici ideile și teoriile cele mai noi.

Eu văd acest proiect ca pe un năvod aruncat peste lume. Recolta va fi următoarea generație de gînditori și oameni de știință ai planetei noastre.

DANIEL C. DENNETT

Cei mai mulți dintre noi, deşi folosim deseori calculatorul, cunoaştem prea puțin despre procesele care au loc în interiorul lui, iar performanțele sale ni se pare că țin mai curînd de vrăjitorie. Evitînd detaliile tehnice şi jargonul de specialitate, Daniel Hillis, unul dintre informaticienii de vîrf ai zilelor noastre, cel care a proiectat calculatorul cel mai rapid din lume, povesteşte în *Maşina care gîndeşte* aventura informaticii. De la expunerea simplă şi intuitivă a funcționării calculatoarelor şi a structurii limbajelor de programare, Hillis ajunge la subiectele arzătoare de azi (creşterea memoriei şi a vitezei de operare) şi la perspectivele tulburătoare ale dezvoltării inteligenței artificiale.

În seria SCIENCE MASTERS au mai apărut:

P. W. Atkins Regatul periodic
William H. Calvin Cum gîndește creierul
Ian Stewart Numerele naturii
Martin Rees Doar șase numere

ISBN 973-50-0141-1

5 948353 000054

O SERIE LANSATĂ CONCOMITENT ÎN PESTE 50 DE ȚĂRI